VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

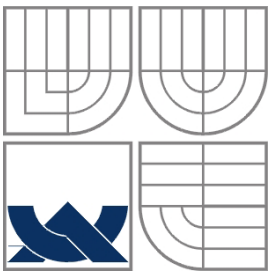
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ALGORITMŮ KOMPRESE DAT S VYUŽITÍM GPU

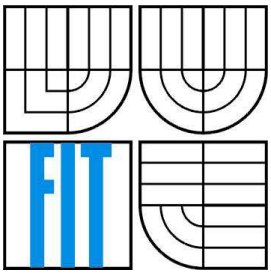
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
PAVEL CACEK

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ALGORITMŮ KOMPRESE DAT S VYUŽITÍM GPU

ACCELERATION OF DATA COMPRESSION ALGORITHMS USING GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PAVEL CACEK
AUTHOR

VEDOUCÍ PRÁCE — ING. VÁCLAV ŠIMEK
SUPERVISOR

BRNO 2013

Abstrakt

Tato bakalářská práce se zabývá možnostmi akcelerace kompresního algoritmu na grafické kartě. Konkrétním zkoumaným algoritmem je kompresní algoritmus JPEG, který se používá pro kompresi obrazových dat. V textu jsou nejprve představeny technologie, pomocí kterých můžeme využívat výpočetní sílu grafických karet. Dále se práce zaměřuje na teoretický popis algoritmu JPEG a následně je popsána jeho implementace pomocí OpenCL a NVIDIA CUDA. Nakonec je provedeno srovnání výkonu těchto GPGPU technologií.

Abstract

This bachelor's thesis is dealing with possibility of acceleration compression algorithm on graphical card. I have studied specifically the compression algorithm JPEG, which is used for compression image data. The text first introduced technology, which give access to us use computational power of graphics cards. The work is also focused on the theoretical description of the JPEG and subsequently is describe its implementation using OpenCL and NVIDIA CUDA. Finally, there is a comparison of performance this GPGPU technologies.

Klíčová slova

NVIDIA CUDA, OpenCL, BMP, RGB, YC_bC_r, DCT, Kvantizace, ZIGZAG přeuspořádání, Huffmanovo kódování, JPEG

Keywords

NVIDIA CUDA, OpenCL, BMP, RGB, YC_bC_r, DCT, Quantization, ZIGZAG reorder, Huffman coding, JPEG

Citace

Cacek Pavel: Akcelerace algoritmů komprese dat s využitím GPU, bakalářská práce, Brno, FIT VUT v Brně, 2013

Akcelerace algoritmů komprese dat s využitím GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Václava Šimka.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Cacek
15. května. 2013

Poděkování

Zde bych rád využil příležitosti k poděkování svému vedoucímu, Ing. Šimkovi za odbornou pomoc a za vedení mé práce.

Obsah

Seznamy

Obrázky:

Tabulky:

Ukázky programového kódu:

1 Úvod

Hlavním polem využití grafických karet bývalo vykreslování počítačové grafiky. S jejich postupným vývojem se u nich dosáhlo velkého výpočetního výkonu, který bylo možné využít v paralelních výpočetních systémech. Tato výhoda dala vznik novému použití grafických karet tzv. GPGPU (General Purpose using of Graphics Processing Unit), což ve volném překladu znamená využití grafického akcelerátoru k obecným výpočtům. Tento nový technologický obor se nejvíce hodí k implementaci algoritmů, které je možné paralelizovat. V dnešní době se GPGPU nejvíce používá pro konverzi, zpracování a kompresi videa.

Tato práce se zabývá využitím grafické karty pro zrychlení tradičních kompresních algoritmů. Pro demonstraci a zpracování byl vybrán algoritmus pro převod obrázku pomocí kompresního ztrátového formátu JPEG. Vstupní obrazová data jsou získávána z oblíbeného rastrového grafického formátu Windows bitmap (.bmp), který vyvinula firma Microsoft. Výstupem je obrázek zkomprimovaný pomocí standardu JPEG, který je uložen v grafickém formátu JFIF (.jpg).

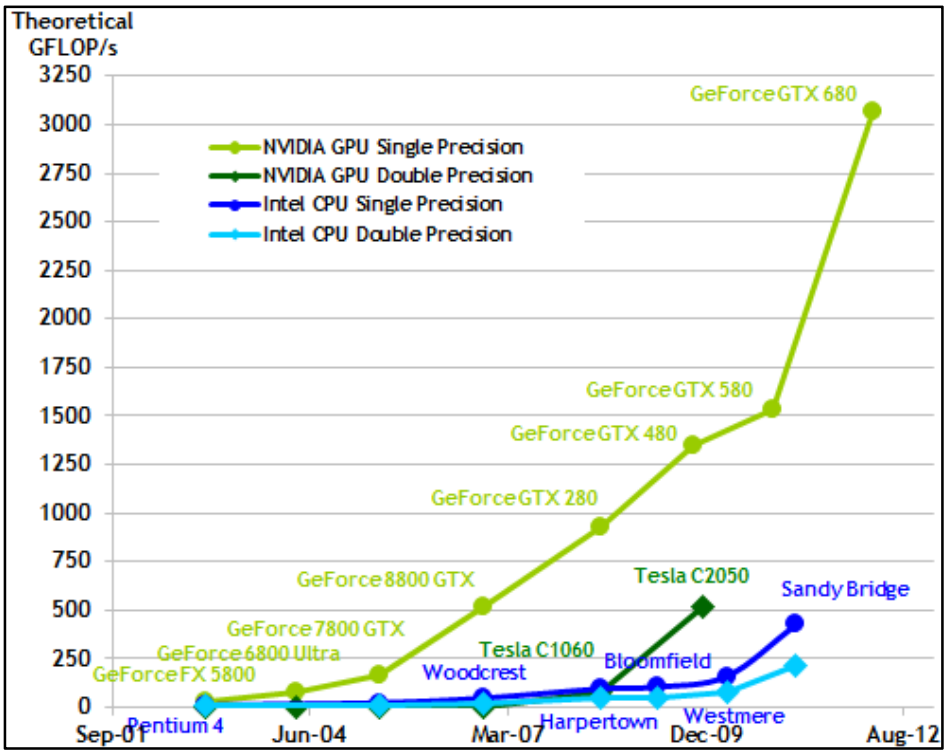
V kapitole **2** budou představeny architektury současných grafických karet a jejich porovnání s architekturami klasických procesorů. V následující kapitole **3** se zaměříme na technologii GPGPU a na programovací prostředí, ve kterých můžeme psát aplikace, které mají být vykonávané na grafických kartách. Následovat bude kapitola **4**, kde se seznámíme s obrázkovými grafickými formáty BMP a JPEG/JFIF.

Kapitola **5** bude blíže popisovat samotnou implementaci programu pro převod BMP obrázků do formátu JPEG, a budou zde detailně rozebrány sekvenční části algoritmu, které budou zpracovávány pomocí technologií NVIDIA CUDA a OpenCL. Následovat bude kapitola **6**, kde se zaměříme na testování a porovnání výkonu obou technologií.

Na závěr budou v kapitole **7** shrnuty a zhodnoceny výsledky této bakalářské práce.

2 Porovnání architektur CPU vs. GPU

Dnešní grafické karty jsou teoreticky několikanásobně výkonnější než klasické procesory [1]. Grafické znázornění výkonu CPU vs. GPU, můžeme vidět na obrázku (Obrázek 1), který je umístěn níže. Tento teoretický výkonový rozdíl je dán tím, že grafické karty jsou zaměřeny více na paralelní výpočty, než výpočetní výkon jednoho vlákna, protože mají mnoho výpočetních jader. Tato výpočetní jádra jsou konstrukčně mnohem menší než jádra klasických procesorů, a také jejich taktovací frekvence bývají nižší, proto je možné jejich architekturu navrhovat a optimalizovat pomocí specializovaných počítačových nástrojů. Tato skutečnost umožňuje jejich levnější a rychlejší vývoj.

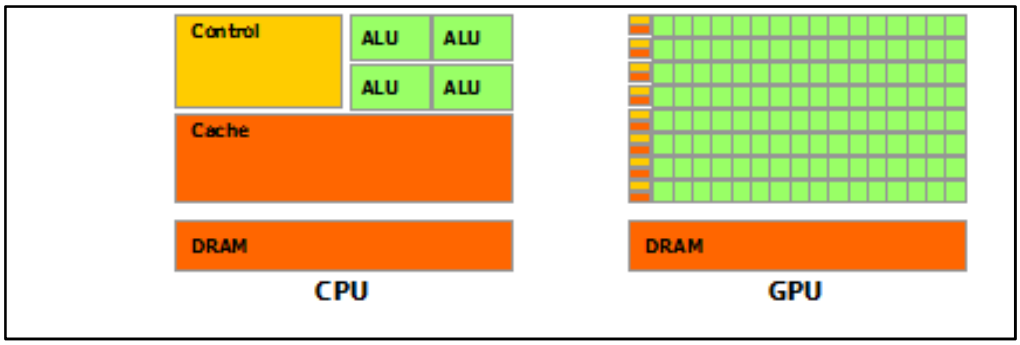


Obrázek 1: Porovnání výkonu současných GPU a CPU v pohyblivé řádové čárce v počtu výpočtů za sekundu (převzato z [1]).

Naproti tomu klasické procesory již narazily na hranici jejich maximální taktovací frekvence (okolo 4GHz), která se při stávajících výrobních technologiích bude již obtížně dále posouvat. Aby procesor bez problémů fungoval, i při takto vysokých taktech, je při jeho vývoji potřeba lidských zásahů a optimalizací, což prodlužuje a prodražuje vývoj procesoru. Proto se výkon dnešních procesorů zvyšuje hlavně přidáváním dalších výpočetních jader (více-jádrové procesory), aktuálně nejčastěji 4 jádrové (v serverech až 16 jádrové).

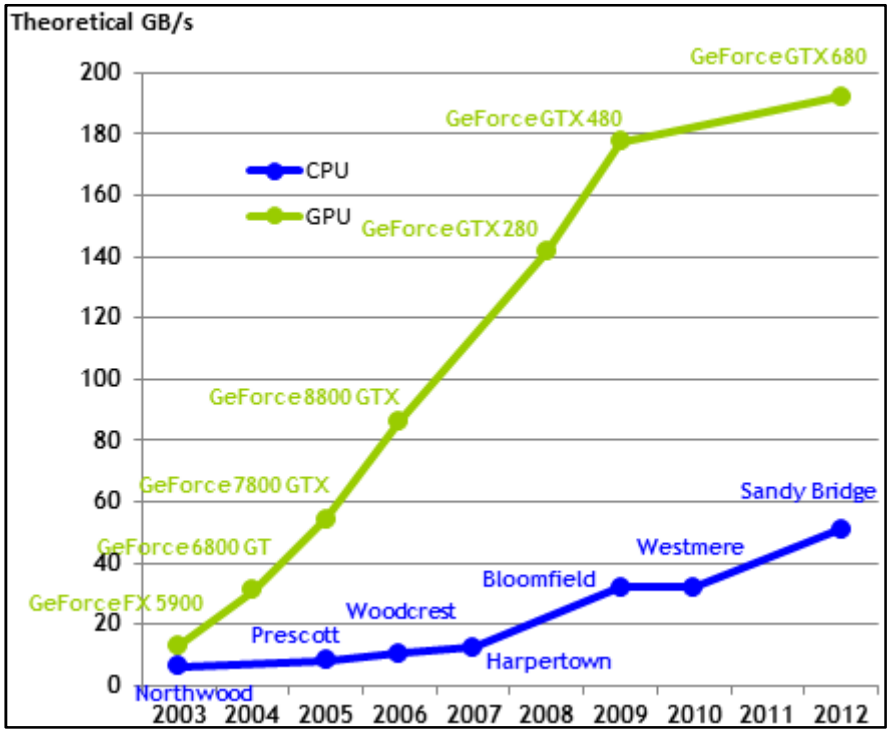
Dnešní procesory i grafiky mají v průměru okolo 1,5 miliard tranzistorů (pro představu současná nejvýkonnější grafická karta Nvidia GK110 Titan má 7,08 miliard tranzistorů).

Je zajímavé prozkoumat rozdíly architektur CPU a GPU, abychom pochopili příčiny toho, proč je jejich výkon odlišný. Schematické zobrazení obou architektur se nachází na následujícím obrázku (Obrázek 3).



Obrázek 3: Porovnání CPU a GPU architertur (převzato z [1]).

Z předcházejícího obrázku vyplývá, že grafické karty jsou nejvhodnější pro řešení problémů, které mohou být vyjádřeny, jako tzv. „data paralelní výpočty“ tzn. stejný programový kód, se provádí nad mnoha datovými prvky. Proto se grafická karta skládá z mnoha SIMD (Simple Instruction, Multiple Data) jednotek (zobrazeny zeleně). Vzhledem k tomu, že se stejný program provádí pro každý datový prvek, snižují se nároky na sofistikované řízení toku vykonávaných instrukcí, a protože se provádí nad mnoha datovými prvky, a má vysokou aritmetickou náročnost, pak latence přístupu do paměti může být skryta pomocí náročnosti výpočtů. Ke grafickým kartám je připojená velice rychlá video RAM, obvykle přes širší sběrnici než klasická RAM k procesoru, tím je vytvořena větší paměťová propustnost video pamětí (Obrázek 2), což do ní zvyšuje přístupovou rychlost. Z povahy



Obrázek 2: Srovnání paměťové propustnosti dnešních CPU a GPU (převzato z [1]).

výpočetních operací na grafické kartě vyplývá, že na cache paměť (oranžová) a blok řízení toku vykonávaných instrukcí (žlutá) u grafických karet nejsou kladeny tak vysoké nároky jako u klasických procesorů.

Klasické procesory, které poskytují určité sofistikované metody řízení toku vykonávaných instrukcí (predikce skoků, načítání argumentů do cache v předstihu, snížení latence přístupu do RAM adt.) musejí mít z tohoto důvodu cache a bloky řízení přizpůsobené svým potřebám (např. současný nejvýkonnější desktopový procesor Intel Core i7-3960X má cache paměť obrovských 15MB).

Aby byl programátor schopen naprogramovat efektivně algoritmus pro běh na grafické kartě, měl by být podrobně obeznámen s její architekturou. V následující kapitole se zaměříme na základní GPGPU technologie a popíšeme si tam i základní strukturu grafické karty.

3 GPGPU

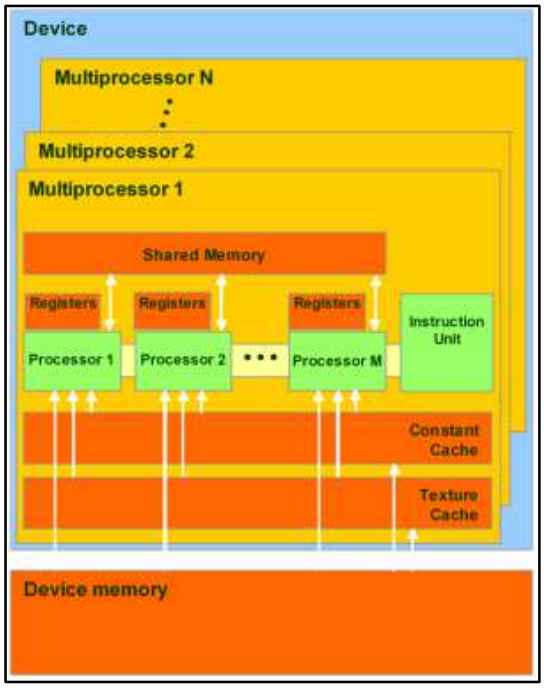
V této kapitole si popíšeme dvě základní programovací rozhraní pro provádění výpočtů na grafické kartě.

3.1 NVIDIA CUDA

V listopadu 2006, firma NVIDIA představila technologii CUDA™ (Compute Unified Device Architecture). Tato hardwarová a softwarová architektura jako první umožnila spouštět programy na grafických kartách firmy NVIDIA. Aplikace na této platformě mohou být psány v programovacích jazycích C/C++, FORTRAN a od letošního roku také v jazyce PYTHON (viz [1]). Důležité je, že podpora této výpočetní platformy je dostupná ve všech hlavních operačních systémech (Microsoft Windows, Linux i Mac OS X). Nevýhoda této grafické výpočetní platformy tkví ve skutečnosti, že aplikace na ní implementované lze spouštět pouze na grafických akcelerátorech firmy NVIDIA.

3.1.1 Hardwarový model

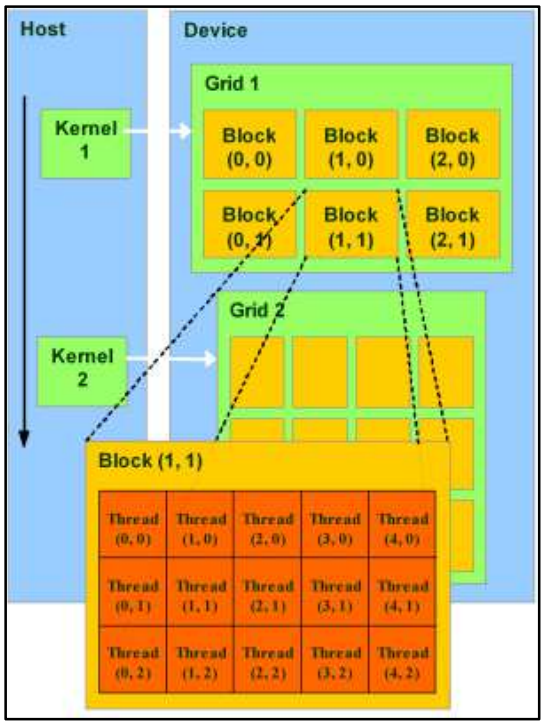
Pro bližší představu si popíšeme vnitřní strukturu grafického čipu NVIDIA s jádrem GT200. Jeho architekturu můžeme vidět na obrázku (Obrázek 4). GPU se skládá z několika tzv. streaming multiprocesorů (SM), jejich počet je modifikovatelný, ale platí pravidlo, že čím je těchto multiprocesorů více, tím je grafická karta výkonnější (architektura GT200 obsahuje na jednom grafickém čipu maximálně 30 streaming multiprocesorů). Každý streaming multiprocesor se skládá z výpočetních procesorů (8 pro architekturu GT200), každý tento procesor obsahuje vlastní paměťové registry, což jsou nejrychlejší paměťové jednotky na grafické kartě. Každý multiprocesor má tzv. Constant cache (kešovaná paměť konstant), která je rychlejší než globální paměť, a proto je výhodné ji na konstanty využívat. Na multiprocesoru se ještě vyskytuje tzv. shared memory (sdílená paměť). Obsah této paměti se sdílí mezi všemi procesory v multiprocesoru a tato paměť je velice rychlá. Na grafickém adaptéru se dále vyskytuje device memory (globální paměť). Globální paměť je sdílená mezi všemi multiprocesory a je to



Obrázek 4: Architektura grafického čipu (převzato z [2]).

největší, ale také nejpomalejší paměť, která se vyskytuje na grafické kartě.

3.1.2 Programový model



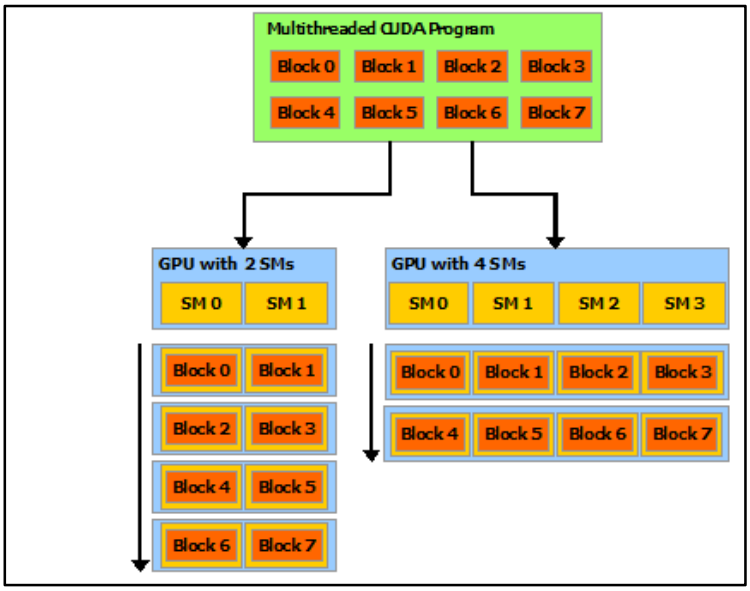
Obrázek 5: Kernel, bloky a vlákna (převzato z [2]).

Hlavní částí programu pro grafickou kartu je tzv. kernel. Tato část programu musí být v programovém kódu CUDA uvozena klíčovým slovem __global__ a je vykonávána v každém spuštěném vlákně. Každé vlákno má svůj index, který ho jednoznačně identifikuje. Vlákna (threads) jsou shlukována do bloků (architektura GT200 dovoluje 512 vláken v jednom bloku) a bloky jsou sdružovány v mřížce (grid) viz (Obrázek 5). Počty vláken a bloků mohou být definovány trojrozměrně, což usnadňuje indexaci a tím i práci s vektory, maticemi a 3D maticemi.

Každý blok se zpracovává na jednom multiprocesoru a jeho vlákna jsou zpracována na jednotlivých procesorech daného multiprocesoru. Samotný výpočet probíhá tak, že ve chvíli, kdy některá vlákna čekají na přenos dat z globální paměti, jiná vlákna, která již data mají načtena, provádějí výpočet. To nám při velkém počtu vláken v bloku zajistí odstínění potřeby extra rychlé globální paměti.

Kernely, které budou na jednotlivých blocích a vláknech zpracovávány, musejí být napsány tak, aby u zpracovávaných dat nezáleželo, v jakém pořadí jsou tato data zpracovávána.

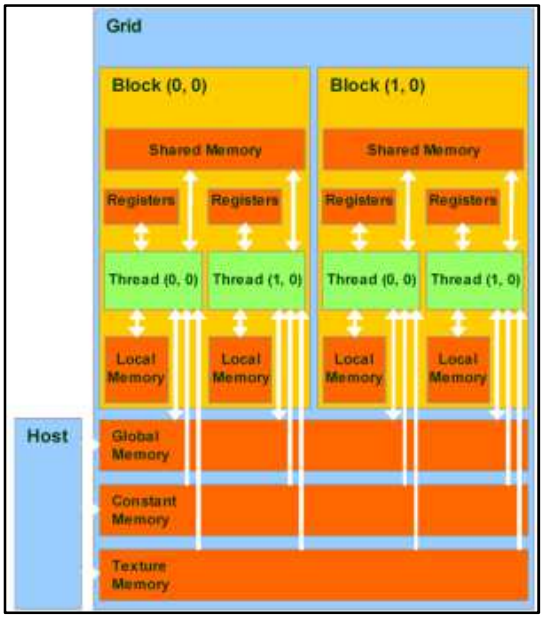
Podle počtu streaming multiprocesorů (SM) grafické karty, na které je běh CUDA programu



Obrázek 6: Přiřazení bloků jednotlivým SM (převzato z [1]).

realizován, si CUDA sama určí, jak budou jednotlivé zpracovávané bloky rozděleny pro zpracovávání na jednotlivých SM. Tuto skutečnost znázorňuje předcházející obrázek (Obrázek 6).

3.1.3 Paměťový model



Obrázek 7: Paměťový model (převzato z [2]).

Při návrhu a implementaci programu, který poběží na GPU, je velice důležité znát paměťový model dané GPGPU technologie. Na obrázku nalevo je vyobrazen paměťový model technologie CUDA (Obrázek 7).

Globální paměť je největší paměť, která se nachází na GPU a je přístupná ze všech multiprocesorů. Její výhodou je velká paměťová propustnost této paměti, ale nevýhodou je její velká latence.

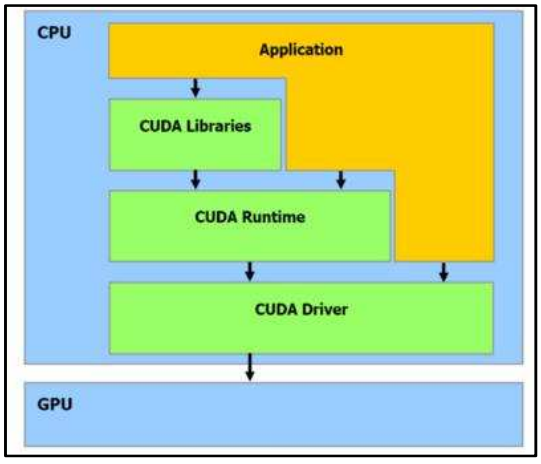
Konstantní paměť se nachází v globální paměti, ale narozdíl od globální paměti je cachovaná, tudíž její čtecí odezvy jsou několikanásobně menší. Z pohledu procesorů na GPU lze použít pouze na čtení, nelze do ní zapisovat.

Texturovací paměť se při GPGPU výpočtech často nevyužívá, má však podobné vlastnosti jako paměť konstantní, ale lze do ní zapisovat.

Sdílená paměť je uložena na multiprocesoru, je velice rychlá a mohou ji využívat všechny vlákna zpracovávaná tímto multiprocesorem.

Každý procesor na multiprocesoru má vlastní registry, což jsou nejrychlejší paměťové jednotky na GPU. Tato paměť bývá velice malá, většinou 8KB na multiprocesor, což vychází kolem 1KB na jeden procesor, proto se s touto pamětí musí šetřit.

3.1.4 Programové API CUDA



Obrázek 8: CUDA API (převzato z [2]).

Jak můžeme vypozorovat z obrázku nalevo (Obrázek 8), na kterém je zobrazen vrstvový model CUDA API (Application Programming Interface), je CUDA aplikace schopná přistupovat ke všem funkcím CUDA API.

API obsahuje knihovny, které obsahují již vytvořené a vysoce optimalizované funkce, které má možnost programátor využít. Mezi tyto knihovny patři

např. CUBLAS (Basic Linear Algebra Subprograms) nebo CUFFT (Fast Fourier Transform).

Samotné programovaní v CUDA C je velice podobné klasickému C/C++. Do tohoto jazyka jsou přidány některé kvantifikátory.

Kvantififátory umístění funkcí:

- __host__ - funkce vykonávaná na procesoru
- __global__ - funkce vykonávaná na GPU a volaná z hlavního programu
- __device__ - funkce vykonávaná na GPU, ale volaná z funkce vykonávané také na GPU

Obdobně se využívají kvantifikátory pro určení, do jakého druhu paměti GPU budou proměnné umístěny. Programové API platformy CUDA je podrobně popsáno v [1].

3.2 OpenCL

OpenCL (Open Compute Language) je framework pro programování paralelních výpočtů na různých hardwarových platformách. OpenCL není na rozdíl od dříve zmiňované NVIDIA CUDA závislý na hardwaru jedné firmy. Programy, které jsou napsané v OpenCL, jsou schopné pracovat na univerzálních procesorech, grafických kartách, ale třeba i na procesorech architektury CELL nebo signálních procesorech DSP.

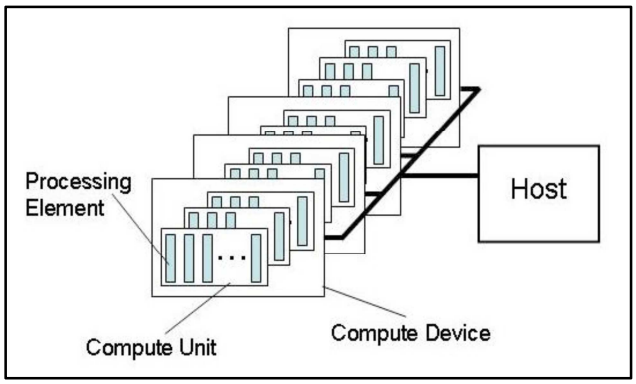
Za vývojem a standardizací OpenCL stojí konsorcium Khronos Group, které také stojí za vývojem např. OpenGL. Skupina Khronos se skládá z firem, jako jsou AMD, Apple, IBM, Intel, NVIDIA, Texas Instrument, Sony a Toshiba. OpenCL je otevřený standard, který byl poprvé představen v roce 2008. Aktuálně se nachází ve verzi 1.2. Více viz [3] a [4].

Programování pomocí OpenCL zahrnuje naprogramování dvou částí. Zaprvé tzv. hostovské části, která je standardně vykonávána na klasickém procesoru a dále částí kernelu, který je vykonáván na vybraném zařízení, které podporuje technologii OpenCL.

Pro základní popis programové architektury OpenCL si jej můžeme rozdělit do hierarchicky uspořádaných modelů:

- Platformový model
- Paměťový model
- Vykonávací model
- Programovací model

3.2.1 Platformový model



Obrázek 9: Platformový model (převzato z [4]).

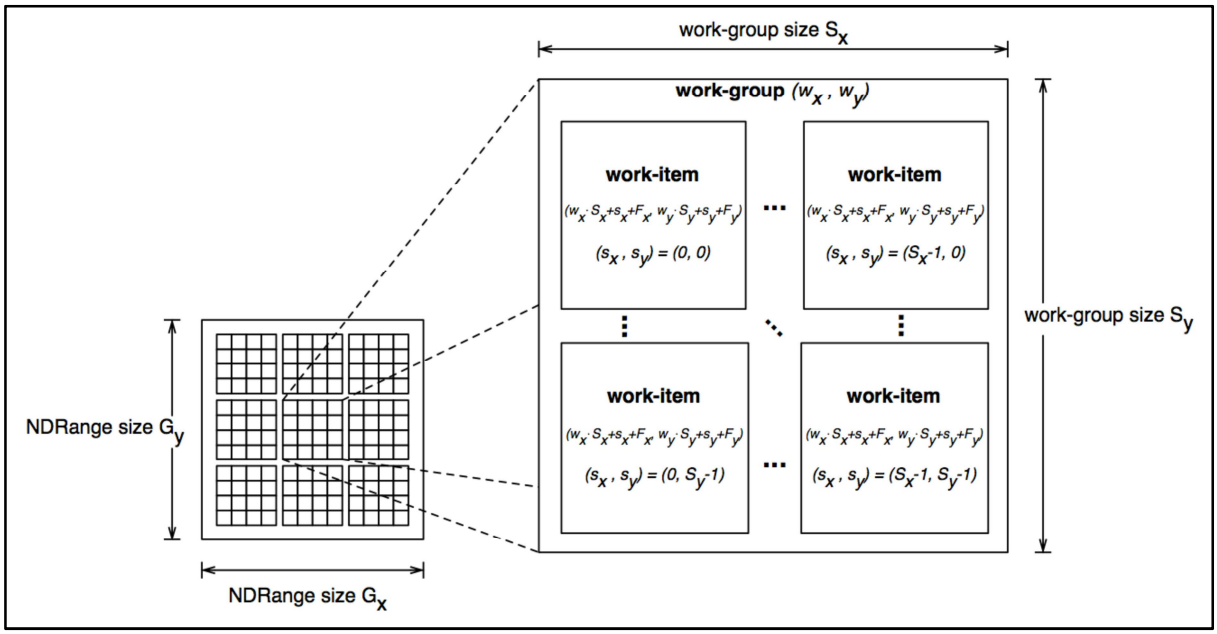
Platformový model zobrazený na obrázku vlevo (Obrázek 9) obsahuje hostitelský systém, na který je připojeno jedno, nebo více OpenCL zařízení. OpenCL zařízení se skládá z jedné nebo více výpočetních jednotek (CUs – compute unit – podobné jako CUDA multiprocesory). Tyto jednotky (CU) dále obsahují několik výpočetních procesorů (PEs – processing elements – stejně jako CUDA procesory).

3.2.2 Vykonávací model

Vykonávací model OpenCL se skládá ze dvou hlavních částí kernelů a hostovského programu. Kernely jsou funkce, které jsou vykonávány na OpenCL zařízeních. Hostovský program definuje

využívaná OpenCL zařízení, stará se o zápis a čtení datových elementů do/z paměti daného OpenCL zařízení a spouští vykonávání kernelových funkcí.

Vykonávací model také definuje, jakým způsobem budou kernely vykonány. Když má být spuštěn výpočet nad daty, je pro daný kernel vytvořen globální indexový prostor. Na každém výpočetním procesoru je vykonávána jedna instance kernelu. Tuto instanci nazýváme work-item a tato pracovní jednotka je jednoznačně identifikována svým globálním indexem (global ID) v rámci



Obrázek 10: Organizace work-item a work-group pomocí NDRange v OpenCL (převzato z [4]).

globálního indexového prostoru. Pracovní jednotky (work-item) mohou být shlukovány do pracovních skupin tzv. work-group, tento princip je znázorněn na výše se nacházejícím obrázku (Obrázek 10). Pracovní skupiny (work-group) jsou zpracovávány na výpočetních jednotkách (CUs) a pracovní jednotky (work-items) jsou vykonávány na výpočetních procesorech (PEs). Každá pracovní skupina má svůj jednoznačný index (work-group ID) a každá pracovní jednotka má navíc v rámci své pracovní skupiny unikátní lokální index (local ID).

To nám určuje, že každá pracovní jednotka může být identifikována dvěma způsoby:

1. pomocí svého global ID
2. pomocí work-group ID a local ID a údaje, jak velké jsou pracovní skupiny (velikost pracovní skupiny je ale omezená, a tato velikost závisí na použitém OpenCL zařízení)

Indexový prostor, který využívá OpenCL se nazývá NDRange. NDRange je N-dimensionální indexový prostor, kde N může nabývat hodnot 1, 2 a 3. To nám dává možnost rozdělit instance kernelů do dimenzí, jaké potřebujeme využít pro zpracovávaná data (vektory, matice a 3D matice).

V hostitelském programu se musí definovat kontext vykonávání kernelů. Kontext zaštiťuje vykonávací model a musí obsahovat tyto zdroje:

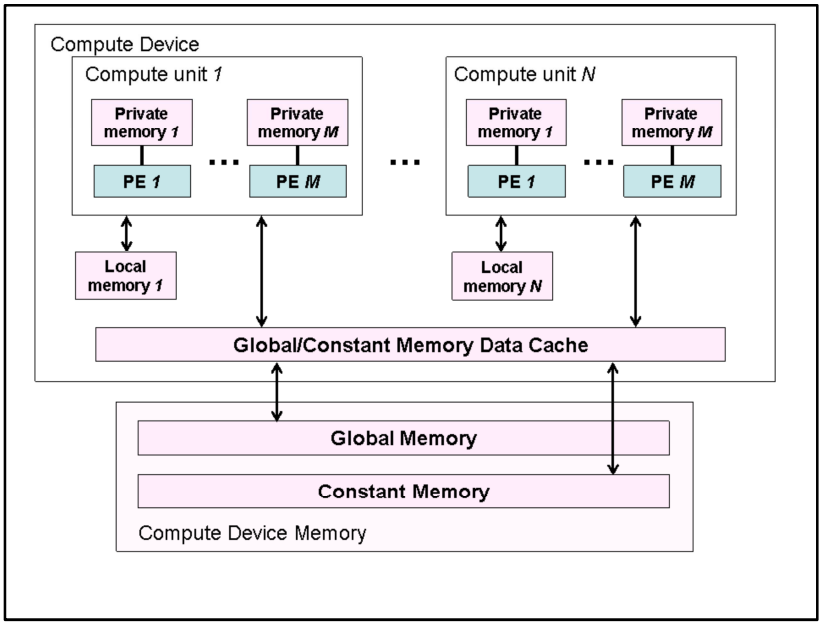
1. Zařízení: OpenCL zařízení, které hostovaný program využívá.
2. Kernely: OpenCL funkce, které poběží na OpenCL zařízení.
3. Programové objekty: Zdrojové a spustitelné kódy programů, které implementují dané kernely.
4. Paměťové objekty: Paměťové objekty, které jsou viditelné jak pro hostitelský systém, tak pro OpenCL zařízení, a ke kterým mohou přistupovat instance kernelů spuštěné na pracovních jednotkách.

Dále je nutné vytvořit fronty příkazů (command-queue), k jednotlivým OpenCL zařízením. Tyto fronty slouží ke vkládání jednotlivých příkazů, které se mají vykonávat na zařízeních. Mezi tyto příkazy patří vykonání kernelu, manipulace s paměťovými objekty a synchronizační příkazy. U příkazových front je možné nastavit, zda mají být příkazy vykonávány v pořadí, v jakém byly zadávány (in-order), nebo se pořadí, v jakém byly zadány, nemusí dodržovat (out-of-order).

3.2.3 Paměťový model

Pracovní jednotky v OpenCL mají přístup do 4 různých paměťových prostorů:

- *Globální paměť:* Tato paměť je přístupná pro čtení i zápis všem pracovním jednotkám ve všech pracovních skupinách, ale není kešovaná.
- *Konstantní paměť:* Část globální paměti, která je přístupná pouze pro čtení a je inicializovaná hostitelským programem. Na grafických kartách firmy NVIDIA je kešovaná, což zvyšuje její rychlost.



Obrázek 11: Konceptuální zobrazení OpenCL zařízení (převzato z [4]).

- *Lokální paměť:* Je to paměťová oblast, která je přístupná v rámci pracovní skupiny (work-group). Tato paměť se nachází přímo na grafickém čipu a je velice rychlá. Využívá se pro alokaci proměnných, které mohou být sdílené mezi pracovními jednotkami (work-items).
- *Privátní paměť:* Tato paměť je vyhrazená pouze pro pracovní jednotku (work-item). Do proměnných, které jsou definované v privátní paměti, nemohou zapisovat jiné pracovní jednotky.

3.2.4 Programovací model

OpenCL vykonávací model podporuje datově paralelní (data parallel) a úlohově paralelní (task parallel) programovací modely, ale také jejich kombinace. Primární model, pro který je OpenCL navrhováno, je datově paralelní.

Datově paralelní programovací model definuje výpočet jako vykonávání stejných sekvencí instrukcí zároveň nad více jednotkami datových objektů. Vykonávací model jednoznačně definuje počty pracovních jednotek (work-items), ale také to, jak do těchto jednotek budou mapovány datové elementy. V nejstriktnějším datově paralelním modelu připadá jedna datová jednotka na jednu pracovní jednotku. V OpenCL je využit datově paralelní model, kde ale striktní mapování jeden na jednoho není nutně vyžadováno. Výhoda datově paralelního modelu je, že pracovní jednotky v rámci pracovní skupiny spolu mohou komunikovat.

Úlohově paralelní model je model, kde jedna instance kernelu je vykonávána nezávisle na indexovém prostoru. Tím pádem je možné spouštět souběžně několik instancí různých kernelů (multitasking). V tomto programovacím modelu spolu ale pracovní jednotky nemohou komunikovat.

V naší demonstrační aplikaci budeme využívat datově paralelní programovací model, protože potřebujeme provádět stejné operace nad mnoha daty.

3.2.5 Programovací jazyk OpenCL C

Pro psaní aplikací využívajících OpenCL se používá programovací jazyk OpenCL C, který je založen na jazyce C podle normy ISO/IEC 9899:1999 – C language specification (dále jen C99). OpenCL C se, ale od C99 liší některými rozšířeními, jako např:

- Vektorové datové typy
- Klíčovými slovy pro určení adresního prostoru
- Klíčovými slovy pro určení, odkud je možné kernel volat, a kde bude vykonáván.
- Kernelové funkce

Ale také má některá omezení vůči C99:

- Nelze využívat ukazatele na funkce
- Rekurze je zakázaná
- Kernelové funkce musejí být typu void (nemají návratové hodnoty)

Kernelové funkce (__kernel) a ostatní programový kód, který má být vykonáván na OpenCL zařízení se zapisuje do souborů s příponou `.cl`. Řídící (hostovský) program se zapisuje v C++ souborech a kernelové funkce jsou z něj spouštěny. Pro každou kernelovou funkci je nutné vytvořit tzv. kernelový objekt. Soubory `.cl` se nekompilují společně s kompletním programem, existují dvě možnosti jejich kompilace.

3.2.5.1 Offline kompilace

V tomto případě se soubory `.cl` překompilují předem do binárního souboru pomocí externího kompilátoru. Řídící program si tento binární soubor při svém běhu pouze načte a z již překompilovaných kernelů si vytvoří kernelové objekty pro dané zařízení, na kterých bude spouštět výpočty.

3.2.5.2 Online kompilace

Při online kompilaci je situace trochu odlišná. Řídící program si při svém vykonávání načte soubor `.cl`, který je pro vybrané OpenCL zařízení překompilován až při běhu aplikace a následně se z tohoto binárního překompilovaného kódu vytvoří kernelové objekty jako v předchozím případě.

4 Využívané grafické formáty

Vstupem vypracovávaného programu jsou obrázky ve formátu Windows Bitmap (.bmp) a výstupem po kompresi jsou obrázky ve formátu JPEG/JFIF (.jpg). Oba tyto obrazové formáty jsou rastrového typu, což znamená, že jsou v obrázku uloženy informace o barvě každého pixelu. U formátu JPEG/JFIF jsou tyto údaje zakódovány pomocí algoritmu JPEG, který si popíšeme v kapitole (4.2).

4.1 BMP (Windows Bitmap)

Grafický formát BMP byl poprvé představen v roce 1988, následně jeho definici firma Microsoft rozšířila a využila ho ve svém 16bitovém grafickém operačním systému Microsoft Windows 3.0.

Velkou výhodou tohoto grafického formátu je, že je ho možné volně používat (není zatížen patentovou ochranou), ale také jeho jednoduchost. Obrázky v tomto formátu jsou ukládány po jednotlivých pixelech. Způsob uložení se liší pouze podle toho, kolik bitů se využívá na uložení jednoho pixelu. Je možné využívat tyto bitové šířky (v závorce je vždy uvedeno kolik barev je možné při dané bitové šířce reprezentovat):

- 1bit na pixel (2)
- 4bity na pixel (16)
- 8bitů na pixel (256)
- 16bitů (65536)
- 24bitů (16777216)

Obrázky uložené ve formátu BMP ve valné míře nevyužívají žádnou metodu komprese obrazových dat. To ale neznamená, že grafický formát BMP kompresi nepodporuje. Existují varianty formátu BMP, které využívají kompresi RLE (run-length encoding). Z důvodu, že komprese se často nevyužívá, jsou většinou obrázky v formátu BMP mnohem paměťově rozměrnější, než stejné obrázky, které jsou ale uloženy ve formátech, které kompresi využívají, jako například jsou PNG a TIFF.

Velkou zvláštností formátu BMP je to, že obrazové pixely jsou ukládány od levého dolního rohu do pravého horního rohu po řádcích, což znesnadňuje jeho sekvenční čtení. Další věcí, na kterou by si měl programátor při využívání formátu BMP dát pozor, je skutečnost, že používá číselný formát little-endian (nejdříve se uloží nejméně významný bajt (LSB) a za něj se uloží ostatní bajty až po nejvíce významný bajt (MSB)), což je rozdíl oproti formátu JPEG/JFIF, který používá číselný formát big-endian (jehož způsob ukládání je opačný). Více o formátu BMP viz [11].

4.1.1 Datová struktura BMP formátu

Vnitřní struktura grafického formátu BMP se skládá z několika částí:

- Bitmap file header (hlavička souboru BMP)
- DIB header (Metainformace o uloženém rastrovém obrazu)
- Color table (Barvová paleta)
- Pixel array (Pole pixelů)

Soubor BMP musí vždy začínat souborovou hlavičkou (Bitmap file header). Tato hlavička obsahuje identifikátor BMP formátu (nejčastěji používaný typ je „BM“, který definovala firma Microsoft), celkovou velikost obrázku a pozici, kde začínají obrazová data (Pixel array).

Za souborovou hlavičkou následuje povinná informační hlavička (DIB header). Těchto informačních hlaviček existuje 7 verzí a rozlišují se podle své velikosti. Tato velikost je uvedena jako první 4 bajty této hlavičky. Pro účely demonstrační aplikace podporuji DIB hlavičku o velikosti 40 B, která se nazývá BITMAPINFOHEADER a je to nejpoužívanější verze informační hlavičky. Jsou v ní obsaženy tyto údaje:

- Velikost DIB hlavičky
- Šířka obrázku zadávaná v pixelech
- Výška obrázku zadávaná v pixelech
- Údaj na kolika bitech je uložen jeden pixel
- Parametr udávající typ použité kompresní metody (BI_RGB – bez komprese, BI_RLE8 a BI_RLE4 – komprese RLE)
- Velikost obrazových dat v bytech
- Horizontální rozlišení výstupního zařízení v pixelech na metr
- Vertikální rozlišení výstupního zařízení v pixelech na metr
- Celkový počet barev v bitmapě (0 znamená maximum)
- Počet barev důležitých pro vykreslení bitmapy (0 znamená, že jsou všechny barvy důležité). Tato vlastnost se využívala dříve, když některá zařízení nebyla schopna zobrazit všechny barvy, pak se tímto údajem určily ty důležité.

Následovat může barvová paleta (nepovinná), která se používá u BMP obrázků, které mají bitovou velikost pixelu 8 bitů a menší. Je to tabulka, která obsahuje tolik barev, kolik je možné při dané bitové šířce použít. Každá jednotlivá barva zabírá 4 B (na každou barevnou složku RGB použit 1B a navíc vždy 1B nulový pro zarovnání). Tyto barvy jsou seřazeny podle indexů od 0 až do $(2^{(\text{bitová šířka pixelu})}-1)$. Pomocí těchto indexů jsou potom barevné hodnoty zapsány v části s obrazovými daty (Pixel array).

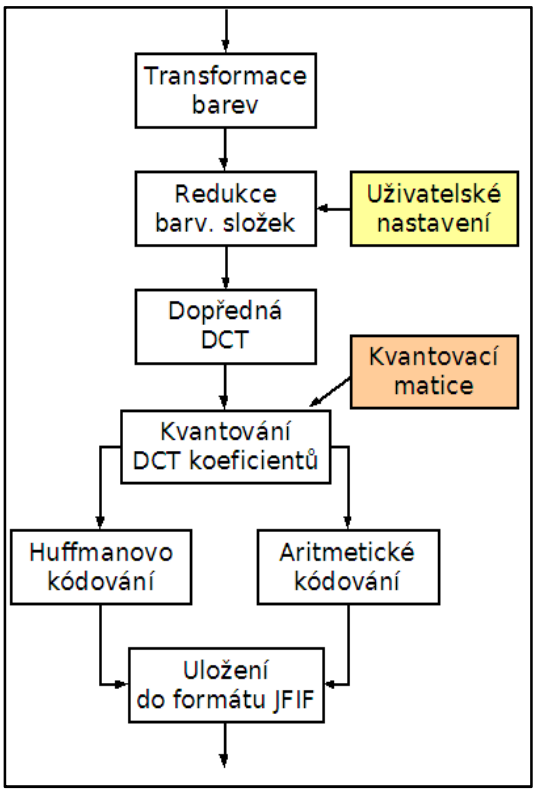
Rastrová data ve formátu BMP (pixely) reprezentují po sobě jdoucí obrazové řádky. Každý řádek obsahuje hodnoty, které zastupují jednotlivé pixely v pořadí zleva doprava. Obrazové řádky ve

formátu BMP musejí být vždy zarovnány na 4 bajty, zbytky pro zarovnání se doplňují nulami. Obrazové řádky se ukládají nezvykle, směrem zdola nahoru.

4.2 JPEG/JFIF

JPEG je sofistikovaná ztrátová kompresní metoda, používaná pro barevné obrázky, ale i pro obrázky, které jsou ukládány ve stupních šedi. Tato metoda využívá ověřeného faktu, že selektivním zanedbáním vybrané (mnohdy nepotřebné) informace, která se nachází v obrázku, můžeme dosáhnout mnohem lepšího komprimačního poměru, než při kompresi bezeztrátové. Samozřejmostí je, že můžeme zanedbat pouze ty informace, které nezpůsobí žádnou viditelnou ztrátu kvality při prohlížení obrázku lidským zrakem. Je důležité si uvědomit, že JPEG nebyl navrhován pro ukládání obrázků, které obsahují malé množství barev. Také jeho použití není příliš vhodné v případě, kdy v obrazových datech jsou konstantní barevné přechody a velmi ostré hrany. Velkým problémem je také písmo. Grafický formát JPEG se nejvíce využívá pro ukládání fotografií a skenovaných dokumentů, ale také v lékařství, pro ukládání rentgenových a ultrazvukových snímků.

Správný název tohoto grafického formátu je JFIF. Název JPEG se pro jeho označení vžil, ale tento název popisuje pouze metodu použitou pro ztrátovou komprimaci rastrového obrázku, která ale nijak nedefinuje, jak bude tento zkomprimovaný obrázek uložen (kompresní algoritmus JPEG se používá i v grafickém formátu TIFF, PDF a i v multimediálním MOV). Formát uložení zkomprimovaných obrazových informací popisuje formát JFIF (používanou koncovkou souborů je .jpg nebo .jpeg, ale dle formátu by bylo správnější použít koncovku .jfif).



Obrázek 12: Kompresní schéma JPEG/JFIF (převzato z [6]).

Název JPEG je zkratka, která vznikla z názvu skupiny, která se od roku 1987 do roku 1991 podílela na jeho vývoji. Tato skupina se nazývala Joint Photographic Experts Group a vznikla spojením členů komise ISO a členů Photographic Experts Group (PEG). Grafický formát JPEG byl jako standard vydán v roce 1993 a tím vzniklo základní kompresní schéma JPEGu více viz [7]. Toto schéma obsahuje fáze, kterými obrázek prochází při své kompresi. Těmito fázemi jsou rozložení obrázku na bloky 8x8 pixelů, transformace barev do modelu YC_bC_r, dopředná diskrétní kosinová transformace (DCT), kvantování koeficientu, seřazení bloku 8x8 podle

ZIGGAG řazení a Huffmannovo nebo aritmetického zakódování (Obrázek 12).
Ve formátu JPEG jsou definovány 4 režimy, které se využívají při kódování/dekódování:

1. Sekvenční kódování
2. Progresivní kódování
3. Bezztrátové kódování
4. Hierarchické kódování

Ze zmíněných režimů se hlavně využívají první dva uvedené. První je nejvyužívanější pro svou relativně malou náročnost na paměť. Druhý, který byl určen pro přenos obrázků po síti, ale je již více náročný na výkon hardwaru, na kterém je zpracovávám. V rámci této práce se budeme zabývat pouze sekvenční implementací komprimačního formátu JPEG a Huffmannovým kódováním.

Aritmetickým kódováním se nebudeme zabývat, protože se dnes často nevyužívá a úspora místa dosažená jeho kódovacími vlastnostmi nedokáže přebít nevýhodu, kterou je velká výpočetní a časová náročnost při jeho realizaci (další nevýhodou je také to, že jeho použití je ovlivněno několika patenty).

4.2.1 Rozdělení obrázku na bloky

Abychom mohli začít sekvenčně komprimovat vstupní obrázek, musíme si ho rozdělit na bloky. Tyto bloky mohou být o velikosti 8x8 či 16x16 pixelů a někdy se nazývají datové jednotky. Výhodou tohoto rozdělení je, že se následně zpracovává blok po bloku, což velice minimalizuje nároky na operační paměť zařízení, na kterém je tato operace vykonávána.

Problém nastává v případě, kdy šířka či výška zpracovávaného obrázku není dělitelná velikostí bloku. Takové obrázky je pro účely zpracování nutno rozšířit vpravo či dolů, aby byla dělitelnost bloky splněna. Toto rozšíření se provádí tak, že nejpravější sloupec a nejspodnější řádek se duplikuje tolikrát, kolikrát je to potřebné v daném rozšiřovaném směru. Matematické operace zmiňované v následujících kapitolách se již provádějí nad samotnými bloky, a na ostatních, ani vedlejších blocích nemají žádnou závislost.

Důležitou věcí je, aby pixely, které se budou komprimovat, měly velikost 8 bitů na jednu barevnou složku. Pokud tuto vlastnost nesplňují, je žádoucí je na tuto bitovou šířku převést.

Pro účely mé demonstrační aplikace používám bloky (datové jednotky) o velikosti 8x8 pixelů.

4.2.2 Transformace barev

Dalším krokem po rozdělení zpracovávaného obrázku na bloky je převedení barevného prostoru zdrojového obrázku (BGB, CMYK či jiný) do barevného prostoru, který používá JPEG. Tímto prostorem je YC_bC_r. Tento barevný model se skládá ze dvou barevných komponent luminance (jasová složka) a chrominance(rozdílová složka). Jasová složka Y nese informaci o světlosti pixelů a

rozdílové složky C_b a C_r nesou informaci o velikosti rozdílu modré a červené oproti jasové složce. Tento barvový prostor se dříve hlavně využíval při přenosu videosignálu, protože umožňoval kompatibilitu s černobílou televizí díky použití jasové složky.

Protože jako vstupní grafický formát používám BMP, ve kterém jsou obrazová data uložena v barvovém prostoru RGB, je nutné převést pixely z modelu RGB do modelu YC_bC_r. K tomuto jsem využil převodních vzorců uvedených v [6].

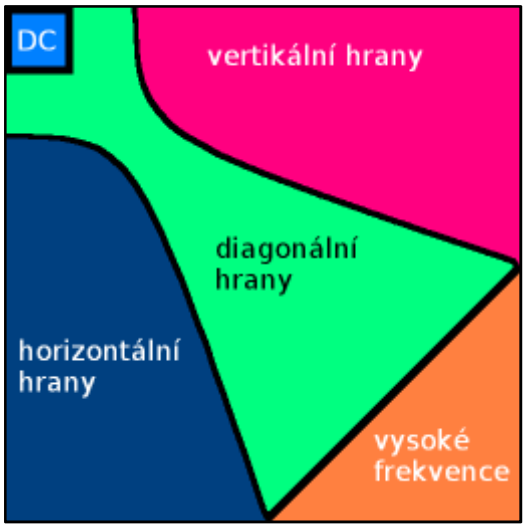
Převod z RGB do YC_bC_r:

$$Y = \left(\frac{77}{256}\right)R + \left(\frac{150}{256}\right)G + \left(\frac{29}{256}\right)B$$
$$C_b = -\left(\frac{44}{256}\right)R - \left(\frac{87}{256}\right)G + \left(\frac{131}{256}\right)B + 128$$
$$C_r = \left(\frac{131}{256}\right)R - \left(\frac{110}{256}\right)G - \left(\frac{21}{256}\right)B + 128$$

Zpětný převod do RGB je možný pomocí těchto vzorců:

$$R = Y + 1{,}371(C_r - 128)$$
$$G = Y - 0{,}698(C_b - 128) - 0{,}336(C_r - 128)$$
$$B = Y + 1{,}732(C_b - 128)$$

4.2.3 Dopředná diskrétní kosinová transformace



Obrázek 13: Výsledná struktura matice pro provedení DCT (převzato z [5]).

Následující fází komprese obrazových dat ve standardu JPEG je diskrétní kosinová transformace. Tato transformace se provádí nad každým blokem, ale pro každou barevnou složku samostatně. Transformace pomocí DCT je bezztrátová (tzn. žádná datová informace se nám při její aplikace neztrácí). Vstupní blok o velikosti 8x8, kde se hodnoty nacházejí v časové doméně, je překonvertován na nový 8x8 blok, který obsahuje koeficienty v doméně frekvenční. Složení tohoto výstupního bloku je znázorněno na obrázku vlevo (Obrázek 13).

Výstupní hodnoty z DCT můžeme rozdělit na dvě kategorie:

- Stejnosměrná složka (DC)
- Střídavá složky (AC)

Stejnosměrná složka je pouze jedna a je to první vypočtený koeficient (v matici je to hodnota nahoře vlevo). Tato stejnosměrná složka ve skutečnosti vyjadřuje průměrnou hodnotu celého zpracovávaného vstupního bloku.

Všechny ostatní hodnoty jsou střídavé kosinové složky a jsou v nich obsaženy informace o hranách a změnách v bloku.

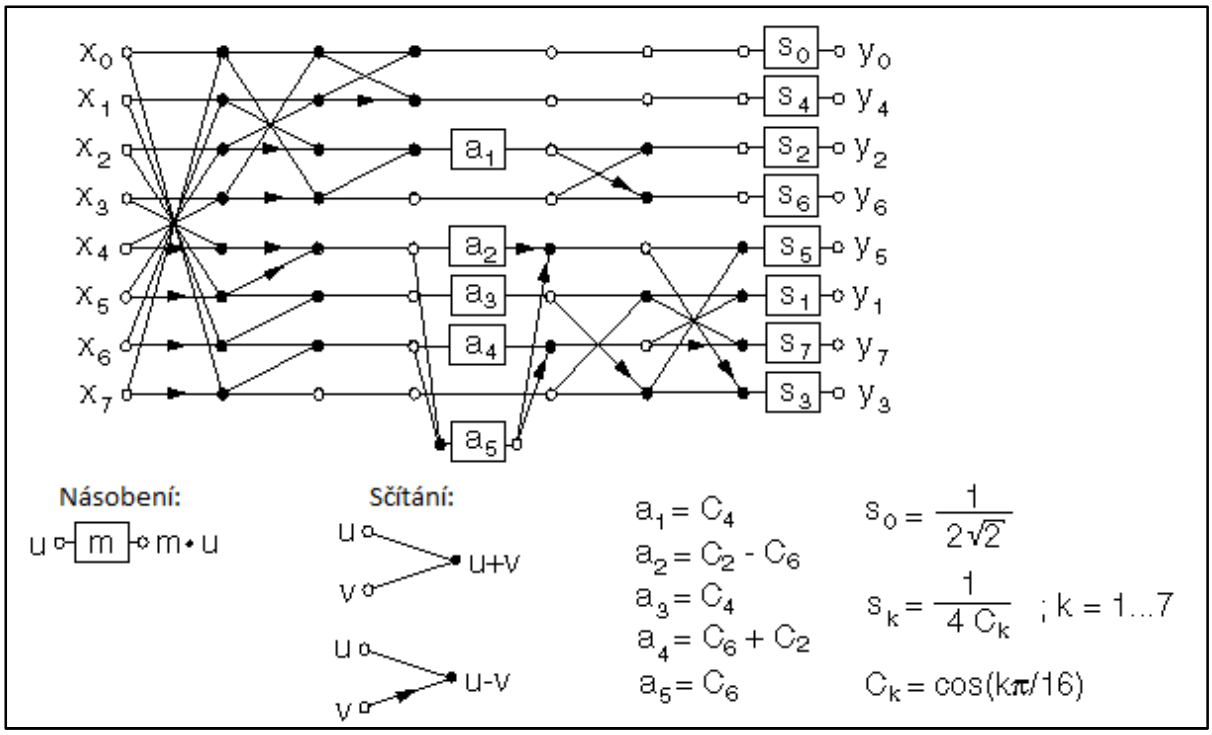
Předtím než nad blokem pixelů může být provedena DCT, je nutné všechny koeficienty v bloku převést z rozsahu [0, 255], který vznikl převodem barevného prostoru, do rozsahu [-128,127] (v tomto rozsahu jsou hodnoty soustředěné kolem nuly, což je pro výpočet DCT žádoucí [6]). To se provede tak, že od každé hodnoty v bloku odečteme číslo 128. Pro výpočet koeficientů JPEG se používá DCT typu II.

Dvoudimenzionální dopřednou diskrétní kosinovou transformaci typu II (2D DCT-II) můžeme vypočítat podle vzorce uvedeného v [6]:

$$G_{ij} = \frac{1}{4} C_i C_j \sum_{x=0}^{7} \sum_{y=0}^{7} p_{xy} \cos\left(\frac{(2x+1)i\pi}{16}\right) \cos\left(\frac{(2y+1)j\pi}{16}\right), \quad kde\ C_f = \begin{cases} \frac{1}{\sqrt{2}}, f = 0 \\ 1, f > 0 \end{cases}$$

Vstupní blok, který je typicky ukládán jako matice, je ve vzorci označen jako p_{xy} a výstupní blok je označován jako G_{ij}. Již ze vzorce je vidět, že výpočet je velice náročný na aritmetický výkon (mnoho násobících operací). Proto byly vyvinuty optimalizované metody pro výpočet DCT.

Jednou z optimalizovaných metod pro výpočet DCT je metoda AA&N DCT 1-D. Tato metoda je popsána v [10] a [12]. Protože tato metoda je jednodimenzionální, je nutné pro zpracovávaný blok spočítat DCT nejprve pro řádky a následně pro sloupce. Každý řádek či sloupec se



Obrázek 14: Dataflow diagram AA&N 1-D DCT s korelačními koeficienty (převzato a upraveno z [10] a [12]).

navíc musí vynásobit korelačními koeficienty. Průběh jednoho řádku či sloupce je podrobně znázorněn na následujícím obrázku (Obrázek 14), na obrázku jsou korelační koeficienty označeny jako S_0 až S_7.

4.2.4 Kvantování koeficientů

Skutečnost, že JPEG je ztrátovým formátem, je zapříčiněna právě kvantováním koeficientů vypočtených diskrétní kosinovou transformací. Kvantizace probíhá tak, že každý blok je celočíselně vydělen koeficienty, které jsou obsaženy v kvantizačních tabulkách. Doporučené kvantizační tabulky pro formát JPEG jsou uvedeny ve standartu [7] a také na obrázku (Obrázek 15). Kvůli rozdílnosti barevných složek se liší kvantizační tabulka pro luminanční (jasovou) složku a pro chrominanční (rozdílovou) složku.

Uspořádání těchto kvantizačních tabulek je založeno na tom, aby pro stejnosměrnou složku a pro střídavé složky o nízkých frekvencích (to jsou ty nejblíže stejnosměrné složce) obsahovaly nízké hodnoty (malé zkreslení po nakvantování), a se zvyšující se frekvencí střídavých složek se tyto hodnoty zvětšují. Častým výsledkem je to, že matice při vysokých frekvencích obsahují nulové hodnoty. Ty jsou velice výhodné při následném Huffmannově kódování. Zvolení správných kvantizačních tabulek má velký vliv na výslednou kvalitu obrazu.

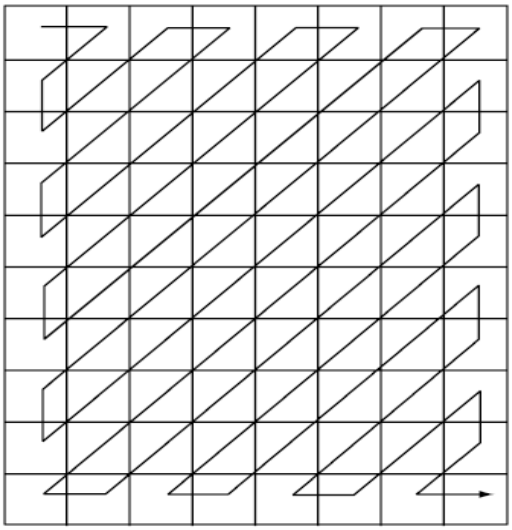
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Luminance

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Chrominance

Obrázek 15: Doporučené kvantizační tabulky pro JPEG (převzato z [6]).



Obrázek 16: Grafické znázornění ZIGZAG řazení (převzato z [6]).

4.2.5 Seřazení koeficientů

Po nakvantování hodnot získáme matici 8x8 DCT koeficientů. Tuto matici je třeba nějak linearizovat (převést do lineárního formátu), abychom nad těmito daty mohli provést Huffmanovo kódování. V JPEGu se na převedení koeficientů do lineární podoby používá způsob nazývaný ZIGZAG, který je znázorněn na obrázku vlevo (Obrázek 16). Tento způsob linearizace si klade za cíl utvořit co nejdelší sekvence nulových koeficientů, které lze použitím Huffmannova kódování velice dobře a úsporně zakódovat.

4.2.6 Huffmannovo kódování

V kódovací části programu jsou zpracovány zlinearizované nakvantované hodnoty z DCT transformace. Tyto hodnoty jsou pro původní 8bitové vzorky v rozsahu [-2048,2047] pro stejnosměrnou složku (DC) a v rozsahu [-1024,1023] pro složku střídavou (AC) [5].

Huffmannovo kódování je založeno na vytvoření slovníku podle četnosti výskytu hodnot. Následně jsou podle četnosti vytvořeny kódy, které reprezentují původní hodnoty. Tyto kódy se vytvářejí tak, aby pro nejčastější hodnoty byly co nejkratší, a čím jsou hodnoty méně časté, tím mohou být kódy delší.

Aby nebylo nutné při každém vytváření obrázku ve formátu JPEG sestavovat Huffmannův slovník četností, jsou ve standardu vytvořeny doporučené Huffmanovy tabulky (viz [7] annex K. 3), které byly vytvořeny po mnoha testech zprůměrováním četnosti hodnot. Ve standardu jsou definovány celkem čtyři Huffmanovy tabulky, protože je výhodné použít různé tabulky pro luminanci a chrominanci. Pro každou tuto složku jsou vytvořeny dvě tabulky, jedna pro DC složku a druhá pro složku AC.

Způsob, jakým jsou kódovány DC koeficienty, je ve formátu JPEG odlišný od způsobu kódování AC koeficientů. U obou způsobů je ale shodné to, že je nutné určit diferenční kategorii pro zpracovávanou hodnotu, tuto hodnotu určíme pomocí tabulky na následující stránce (Tabulka 1). Tato hodnota kategorie nám poslouží jako index do Huffmannovy tabulky pro danou složku, ale také vyjadřuje, na kolika bitech bude uložena vstupní zakódovaná hodnota. Následně jsou v tabulce ještě obsaženy binární posloupnosti, které představují zakódované hodnoty pro dané vstupní rozsahy.

Diferenční kategorie	Rozsah vstupních hodnot	Rozsah zakódovaných hodnot po zakódování v binární formě
0	0	0
1	-1,1	0,1
2	-3,-2,2,3	00,01,10,11
3	-7..-4,4..7	000..011,100..111
4	-15..-8,8..15	0000..0111,1000..1111
5	-31..-16,16..31	00000..01111,10000..11111
6	-63..-32,32..63	000000..011111,100000..111111
7	-127..-64,64-127	0000000..0111111,1000000..1111111
8	-255..128,128..255	00000000..01111111,10000000..11111111
9	-511..-256,256..511	000000000..011111111,100000000..111111111
10	-1023..-512,512..1023	0000000000..0111111111,1000000000..1111111111
11	-2047..-1024,1024-2047	00000000000..011111111111,10000000000..11111111111

Tabulka 1: Diferenční kategorie pro AC i DC koeficienty a jejich zakódované hodnoty pro vstupní hodnoty (převzato z [6] a [7]).

Princip zakódování je takový, že nejdříve se určí pro danou hodnotu její diferenční kategorie, která určí index do Huffmannových tabulek. Následně se do výstupního souboru zapíše kódovaná hodnota dané kategorie a hned za ní se umístí zakódovaná hodnota pro danou vstupní hodnotu, která je daná předešlou tabulkou (Tabulka 1). Základní způsob kódování DC a AC složek je podobný, ale v některých podrobnostech se liší, což bude popsáno dále.

4.2.6.1 Kódování DC složek

Protože hodnota DC složky zastupuje v podstatě průměrnou hodnotou celého zpracovávaného bloku, je časté, že hodnota DC složky je velká. Proto se zavedl způsob, kdy se od DC hodnoty zpracovávaného bloku odečte DC hodnota předešlého bloku stejné barvy. Pokud se jedná o první zpracovávaný blok, je odečtena 0. Tento princip je znázorněn následujícím vzorcem a také příkladem převzatým z [6], který bude následovat.

$$DIFF = DC - PRED$$

Příklad:
Mějme tyto tři DC koeficienty jdoucí za sebou 1118,1114 a 1119. První koeficient 1118 se zakóduje tak, že se od jeho hodnoty odečte 0, tím pádem se jeho hodnota nezmění a zakóduje se jako 1118. Pro zakódování druhého koeficientu 1114 je nutné od něj odečíst předešlou hodnotu, kterou je 1118, takže po odečtení nám vznikne hodnota -4, která se zakóduje. Následující třetí hodnota se zakóduje obdobným způsobem, odečte se 1119 - 1114 čímž vznikne hodnota 5 a ta se následně také zakóduje.

Na tomto příkladu můžeme vidět výhodu tohoto principu, protože výsledné kódované koeficienty jsou mnohem menší, a tím pádem kódovací posloupnosti budou kratší, a tím se zlepší i vlastnosti celkové komprimace.

4.2.6.2 Kódování AC složek

Při kódování AC složek se princip, který byl použit u DC složek, nevyužívá. Zde se využívá jiného principu. Kódují se pouze nenulové hodnoty. Před každou nenulovou hodnotou se spočítá počet nulových hodnot a podle tohoto počtu a diferenční kategorie kódované AC hodnoty se určí z Huffmannovy tabulky prefix. Toto je důvod, že Huffmannovy tabulky pro AC koeficienty jsou mnohem vetší než pro DC koeficienty. Pokud nastane situace, že před kódovanou hodnotou se vyskytuje více než 16 nulových hodnot, pak se každých 16 nulových hodnot zakóduje speciální kódovací hodnotou ZRL a od počtu nul před kódovanou hodnotou se odečte číslo 16, toto opakujeme do té doby, než je před kódovanou hodnotou méně než 16 nulových hodnot. Dále se využívá ještě jednoho principu. Pokud za nenulovou hodnotou v daném bloku následují do konce bloku již samé nulové hodnoty, daný blok se již dále nezpracovává a zakóduje se speciální hodnota EOB (End Of Block), která nám říká, že až do konce bloku následují již samé nulové hodnoty. Tento princip je velice často používaný, protože ZIGZAG řazení na konec sekvence umístí AC koeficienty, které mají vysoké frekvence a ty jsou ve většině případů po nakvantování nulové.

4.2.7 Uložení do grafického formátu JFIF

Jak již bylo zmiňováno dříve, JPEG není grafickým souborovým formátem, je pouze grafickou kompresní metodou. Na uložení obrazové informace používáme grafický souborový formát JFIF. Soubor ve formátu JFIF se skládá z více částí (segmentů) a tyto části jsou oddělovány pomocí tzv. značek. Ve formátu JFIF se používá číslicové uspořádání big endian (byte s nejvyšší váhou (MSB) je na prvním místě a za ním následují ostatní byty až po bajt s nejnižší váhou (LSB)).

Některé nejčastěji používané značky jsou uvedeny v tabulce na další stránce (Tabulka 2). Kompletní specifikace a formát dat používaný v segmentech ve formátu JFIF je možné nalézt ve standardu [7] v části Annex B.

Formát většiny segmentů je takový, že je uvedena značka segmentu, a za touto značkou je na 2 bajtech uvedena velikost celého segmentu, bez velikosti samotné značky. Za velikostí již následují samotné parametry daného segmentu.

Značka	Název	Hodnota	Význam
SOI	Start Of Image	0xffd8	Značka určující začátek souboru
APP0	Application Marker	0xffe0	Značka identifikující soubor JFIF
DQT	Define Quantization Table	0xffdb	Značka určující začátek segmentu koeficientů kvantizační tabulky
SOF0	Start Of Frame 0	0xffc0	Segment obsahující specifikace tvaru obrazových dat
DHT	Define Huffman Table	0xffc4	Značka určující segment kódových slov Huffmannova kódování
SOS	Start Of Scan	0xffda	Značka určující začátek segmentu kódovaných obrazových dat
COM	Comment	0xfffe	Značka obsahující jednoduchý textový komentář
EOI	End Of Image	0xffd9	Značka určující konec souboru

Tabulka 2: Základní značky používané v JFIF (převzato z [7]).

5 Návrh a implementace

V této kapitole bude podrobně popsána implementace kompresního algoritmu JPEG pomocí grafické karty. Budou zde podrobně rozebrány sekvenční části tohoto algoritmu, které je možné zpracovávat na GPU.

Aplikace, která byla pro účely této práce vyvíjena, se nazývá `jpeg_gpu` a je založena na kompresním algoritmu JPEG. Tato komprese je v demonstrační aplikaci realizována ve čtyřech variantách výpočtu. První varianta pojmenovaná DEF (default) je implementována hlavně kvůli porovnání výkonu s dalšími variantami. Implementace v této variantě je provedena v C++ a výpočet běží pouze v jednom výpočetním vlákně, tudíž tato varianta znázorňuje výkon pouze jednoho logického jádra klasického procesoru. Druhou variantou je CUDA. Tato implementace kompresního algoritmu je provedena pomocí frameworku NVIDIA CUDA a běží na grafické kartě. Třetí a čtvrtá možnost spuštění výpočtu jsou si podobné, jejich společnou vlastností je to, že samotný výpočet je realizován pomocí technologie OpenCL. Liší se pouze v té skutečnosti, zda je výpočet uskutečněn na grafické kartě, nebo na klasickém procesoru. Aplikace byla vyvíjena na operačním systému Windows 8 ve vývojovém prostředí Microsoft Visual Studio 2010 v jeho 32-bitové verzi.

Rozebereme-li kompresní algoritmus JPEG, zjistíme, že jeho sekvenční části, které lze zpracovávat najednou (ve své podstatě paralelně, což je základ pro data-paralelní GPGPU) jsou transformace barevného modelu, diskrétní kosinová transformace a kvantizace koeficientů DCT. Tyto jednotlivé části budou popsány dále.

Program je napsaný v jazyce C++ a skládá se z jednotlivých modulů. Hlavní částí programu je modul `Main.cpp`, který se stará o celkové řízení běhu programu. Důležitou součástí celého programu je hlavičkový soubor `MyHeader.h`, který využívají všechny moduly. Tento hlavičkový soubor obsahuje includy na použité hlavičkové soubory, definuje nové využívané datové typy, obsahuje definice konstant využívaných v programu a deklarace globálních funkcí. Následně se aplikace skládá z dalších modulů. `MyParser.cpp` se stará o načtení a zpracování vstupní bitmapy, `MySaver.cpp` obstarává vytvoření výstupního obrázku a Huffmannovo kódování. Samotná komprimační část je implementována ve třech modulech podle toho, jakým způsobem mají být data zpracovávána. V modulu `MyCPUFunc.cpp` jsou implementovány funkce zpracování barev, DCT a kvantování pomocí jazyka C++. Modul `MyGPUFuncCUDA.cu` obsahuje tyto funkce implementované pomocí technologie CUDA a v `MyGPUFuncCL.cpp` jsou tyto funkce realizovány pomocí OpenCL. V následujícím textu budou tyto moduly podrobněji rozebrány.

Aplikace zpracovává vstupní obrázek ve dvou fázích. V první fázi načte vstupní obrázek a rozdělí ho do bloků 8x8 pixelů. Samotné pixely jsou v jednotlivých blocích navíc ještě přeindexovány podle ZIGZAG řazení. Takto uspořádané bloky jsou uloženy do dočasného souboru. Ve druhé fázi jsou bloky čteny z dočasného souboru. Počet bloků, které jsou najednou načítány, je závislý na velikosti

paměti, ve které budou datové elementy při komprimaci uloženy. Důvodem toho, že vstupní obrázek musí být zpracováván v těchto dvou fázích je to, že pixely jsou v bitmapových obrázcích uloženy po řádcích od spodního až po vrchní, ale do výstupního grafického formátu JPEG/JFIF jsou ukládány po blocích od shora dolů. Proto je nutné seřadit pixely od shora dolů do bloků do dočasného souboru, aby bylo následně možné číst bloky sekvenčně, což nám zajistí i to, že je možné regulovat počet najednou načítaných (zpracovávaných) bloků.

5.1 Využívané datové typy a struktury

Všechny ručně definované datové typy jsou uvedeny v hlavičkovém souboru `MyHeader.h`. Důležitým datovým typem, který je využíván v rámci celé aplikace, je datová struktura `image` viz (Kód 1).

```
typedef struct _image{
  header my_header;           //základní hlavička bmp
  dib_header my_dib_header;   //informační hlavička bmp
  blocks_RGB my_rgb;          //vstupní bloky RGB (unsigned char*)
  blocks_QT my_blocks;        //bloky pro DCT YCbCr (short*)
  Qt_table my_qt_table;       //Kvantizacni tabulky
  Ht_table my_ht_table;       //Huffmannovy tabulky
  dc my_dc_coef;              //hodnoty DC složky předchozích bloku
} image;
```

Kód 1: Datová struktura image.

Struktura `image` je datovým typem základní globální proměnné `my_image`, která se využívá v rámci celé aplikace a sdružuje v sobě údaje o vstupním obrázku, ale také datové struktury obsahující pole obrazových pixelů, kvantizační tabulky a Huffmannovy tabulky.

Struktury `header` a `dib_header` obsahují všechny údaje ze vstupního bitmapového obrázku, jako jsou umístění obrázku, výška a šířka obrázku v pixelech, hodnota udávající na kolika bitech je uložen jeden pixel, typ komprese vstupního obrázku atd. Datová struktura `blocks_RGB`, kterou můžeme vidět níže (Kód 2), se skládá ze třech ukazatelů na pole typu `unsigned char` (8 bitů). Do těchto polí jsou načtena obrazová data, která jsou již rozdělena do bloků 8x8 pixelů. Tato struktura se používá jako vstup dat do funkce, která převádí barevné modely.

```
typedef struct _bRGB{
      unsigned char *Red;
      unsigned char *Green;
      unsigned char *Blue;
} blocks_RGB;
```

Kód 2: Datová struktura blocks_RGB.

Struktura `blocks_QT` je struktuře `blocks_RGB` velice podobná, jenom s tím rozdílem, že obsahuje ukazatele na pole typu `short` (16 bitů). Datový typ `short` je nutný, protože se nad touto

strukturou provádí diskrétní kosinová transformace, která pro 8-bitové vstupní vzorky, které tato aplikace využívá, dává výsledky na 12-bitech, což spadá do rozsahu datového typu `short`.

Následně jsou ve struktuře `image`, datové struktury `Qt_table` a `Ht_table`. Struktura `Qt_table` obsahuje kvantovací tabulky pro jasovou i rozdílovou složku obrázku. Tyto tabulky jsou vynásobeny korelačními koeficienty pro 1-D AA&N DCT, aby ubylo výpočtů při samotné DCT a kvantizaci. V `Ht_table` jsou obsaženy Huffmannovy tabulky AC a DC pro jasovou i rozdílovou složku, a také tabulka diferenčních kategorií. Nakonec ve struktuře `dc` je obsažena proměnná pro každou barevnou složku (YC_bC_r = 3 proměnné), do které se ukládá při Huffmannově kódování DC složek barva předcházejícího bloku.

V modulu MyHeader.h jsou také uvedeny standardní kvantovací tabulky ([7], annex K. 1), předloha pro vygenerování Huffmannových tabulek ([7], annex K.3) a mapovací funkce pro ZIGZAG řazení.

5.2 Řízení běhu aplikace

O kompletní řízení běhu aplikace se stará funkce `main()` obsažená v modulu `Main.cpp`. Po spuštění aplikace jsou nejdříve vyhodnoceny vstupní argumenty. Pokud nejsou zadány žádné argumenty, nebo je zadán jediný argument `--help` na výstup vypsána nápověda. Jestliže není počet argumentů správný, nebo nejsou argumenty validní, je vypsáno chybové hlášení. Pokud jsou argumenty v pořádku, je zaznamenán způsob, jakým bude komprimace probíhat, a také je uloženo umístění vstupního obrázku.

Jakmile jsou zpracovány spouštěcí argumenty, je přistoupeno k fázi parsování vstupního obrázku, která je detailně popsána v podkapitole 5.2.1. Následně po ukončení předchozí fáze je podle vstupního argumentu udávajícího způsob realizace komprimačního algoritmu JPEG spuštěn jeho běh nad vstupními daty, pomocí zadané technologie (CUDA, OpenCL, sekvenčně – jednovláknově).

Po vyhotovení komprimační fáze je dealokována využívaná paměť a aplikace je ukončena.

5.2.1 Načtení a přetransformování vstupního obrázku

Funkce pro načítání a zpracování vstupního bitmapového obrázku jsou implementovány v modulu `MyParser.cpp`. Tento modul si na počátku načte souborovou i informační hlavičku BMP souboru a ze získaných dat určí, zda je daný obrázek programem podporován (podporovány jsou BMP obrázky s informační hlavičkou typu BITMAPINFOHEADER, což je nejčastější varianta, se kterou se můžeme setkat).

Z důvodu, že v bitmapových obrázcích jsou obrazové informace uloženy po řádcích od spodního po horní, jak již bylo zmíněno v podkapitole 4.1.1, je výhodné před samotnou komprimací obrazová data přeuspořádat do bloků 8x8 pixelů, které jsou seřazeny od levého horního rohu obrázku

po jeho pravý spodní roh. Proto si naalokuji pole (typu `unsigned char`) pro každou barevnou složku. Velikost tohoto pole je dána šířkou a výškou obrazových dat rozšířených na násobky 8.

Následně jsou obrazová data ze vstupního obrázku čtena po řádcích a pro každý pixel je vypočten index, který určí pozici, kam je tento pixel uložen. Navíc jsou jednotlivé bloky již uspořádány podle ZIGZAG řazení, aby se snížila paměťová náročnost a odpadla fáze řazení. Tato skutečnost ovlivňuje pouze to, že k jednotlivým prvkům v bloku je nutné přistupovat pomocí indexační ZIGZAG matice.

U obrazových dat je při rozdělování na bloky důležité, aby výška a šířka obrázku v pixelech byla dělitelná velikostí bloku. Pokud obrázek toto pravidlo nesplňuje, je jeho výška či šířka rozšířena na násobky velikosti bloku. Způsob rozšíření je popsán v části 4.2.1.

Když je takto načten a přeuspořádán celý vstupní obrázek, je ve vstupní složce vytvořen dočasný soubor, jehož název se shoduje s názvem vstupního obrázku, ale má koncovku `.tmp`. Do tohoto souboru jsou zapsány všechny bloky vzniklé přeuspořádáním vstupního obrázku. Bloky jsou zapisované jeden po druhém v tomto pořadí: první blok R, první blok G, první blok B, druhý blok R a takto je do dočasného souboru zapsán celý obrázek. Následně jsou tato pole dealokována a fáze zpracování vstupního obrázku tímto končí.

Vytvořená kompresní aplikace disponuje jedním omezením, kterým je ta skutečnost, že je schopna zpracovávat pouze vstupní obrázky do velikosti cca 1400MB. Tato skutečnost je zapříčiněna tím, že vývoj probíhal pomocí vývojového prostředí Microsoft Visual Studio 2010 v 32-bitové verzi, ve které vzhledem k architektuře PC není možné alokovat více operační paměti. Tato paměť je potřebná ve fázi předzpracování obrázku, jak bylo popsáno v této části, kde je nutné celý obrázek přerovnat do jednotlivých bloků.

5.2.2 Určení počtu najednou zpracovávaných obrazových bloků

Po úspěšném dokončení části aplikace, kde je vstupní obrázek načten, seřazen a uložen do dočasného souboru, se přistupuje k hlavní části aplikace, kterou je samotná komprese vstupních obrazových dat. V této části se způsob výpočtu liší podle zvolené techniky (CUDA, OpenCL, sekvenčně – jednovláknově), jakou bude výpočet probíhat. Protože obrázek není zpracováván celý najednou, ale po jednotlivých blocích, je důležité si určit, kolik bloků může být najednou načteno do paměti.

Skutečnost, kolik bloků bude načítáno z dočasného souboru, také ovlivňují datové typy použité při kompresi a velikost paměti, kterou je možné alokovat. Při komprimaci se využívají dva datové elementy z globální struktury `my_image`, těmito elementy jsou `my_rgb` a `my_blocks`. Struktura `my_rgb` je typu blocks_RGB tudíž obsahuje tři pole pro barevné složky RGB (`unsigned char`) tudíž na jeden blok je velikost této struktury 64*3*velikost(`unsigned char`, což je 1B). Druhá struktura `my_blocks` je typu `blocks_Qt` a také obsahuje tři pole, ale tentokrát pro barevné složky YC_bC_r (`short`). Takže velikost jednoho bloku obrázku v této struktuře je

64*3*velikost(`short`, což je 2B). Obě tyto struktury musí být naalokovány na uložení stejného počtu bloků, protože do `my_rgb` se ukládají vstupní bloky přímo z dočasného souboru a dále se tato struktura použije jako vstup do zpracování barev a struktura `my_blocks` se využívá jako výstup ze zpracování barev.

Z předešlého odstavce vyplývá, že na jeden barevný pixel jedné barevné složky potřebujeme 3B, z čehož určíme, že na jeden pixel, který se skládá ze tří barevných složek, potřebujeme 9B. Takže na uložení jednoho zpracovávaného bloku (8x8 pixelů) potřebujeme 64*9B, což je 576B a touto hodnotou se musíme řídit, když určujeme kolik bloků budeme najednou zpracovávat, abychom byli schopni tuto určenou kapacitu na tyto bloky naalokovat.

Pokud kompresi zpracováváme pomocí grafické karty, musíme být schopni tuto kapacitu naalokovat také v její video paměti, protože grafická karta může přistupovat pouze ke své paměti (není schopna přistoupit k RAM). Před samotným výpočtem vstupní data do této video paměti nakopírujeme z RAM paměti a po ukončení výpočtu výstupní data nakopírujeme zpátky do RAM.

5.2.2.1 Sekvenční – jednovláknová implementace

U jednovláknové implementace není množství najednou zpracovávaných bloků až tolik důležité, protože se najednou zpracovává pouze jeden blok. Aby nebylo nutné každý blok zvlášť načítat ze souboru, ale také kvůli snížení velikost používané paměti, byl zvolen po několika testech kompromis načtení maximálně 180MB (327680 bloků), jejich následné zpracování a případné načtení dalších bloků.

5.2.2.2 CUDA implementace

Zvolení počtu najednou zpracovávaných obrazových bloků při implementaci v CUDA je již náročnější. Je nutné zjistit kolik má daná CUDA grafická karta volné paměti. Dále je důležité, kolik může být najednou maximálně použito výpočetních bloků (tzv. maxgrid), a také kolik může být na výpočetním bloku spuštěno vláken. Tyto údaje jsou důležité, protože ve fázi převodu barev je každý pixel v bloku zpracováván samostatným vláknem. Z toho vyplývá, že na zpracování jednoho bloku je potřeba 64 vláken. Takže počet najednou zpracovávaných obrazových bloků zjistíme tak, že vynásobíme (max-cuda-bloků*max-cuda-vlaken-v-bloku)/64. Přitom víme, že na jeden zpracovávaný blok je potřeba 576B paměti, takže je případně nutné počet najednou zpracovávaných obrazových bloků snížit, aby se nepřesáhla volná paměť na CUDA zařízení.

5.2.2.3 OpenCL implementace

Určení počtu zpracovávaných bloků v OpenCL je podobné jako u CUDA. Je nutné zjistit kolik je možné na vybraném OpenCL zařízení spustit pracovních jednotek (work-items) v pracovní skupině (work-group). Počet najednou spuštěných pracovních skupin (work-groups) není omezen. Důležitá je hodnota maximálního množství naalokované paměti na daném OpenCL zařízení. Z této hodnoty

určíme velikost paměti, kterou budeme alokovat a také počet najednou zpracovávaných obrazových bloků.

5.2.3 Inicializace kompresních tabulek a vytvoření výstupního souboru

Před samotnou kompresí vstupních dat je potřeba nainicializovat kvantizační tabulky, Huffmannovy tabulky a tabulku diferenčních kategorií, která se používá při Huffmannově kódování. Dále je nutné vytvořit výstupní soubor ve formátu JFIF, zapsat do něho hlavičková data a připravit ho k zápisu pixelů. Všechny tyto zmiňované operace se provedou po zavolání funkce `new_jpeg_init()`, která je definována v modulu `MySaver.cpp`.

Po spuštění této funkce se nejdříve vytvoří výstupní soubor. Jméno výstupního souboru bude shodné se jménem vstupního souboru, ale lišit se bude koncovkou, která v případě výstupního souboru bude `.jpg`.

Následně jsou vytvořeny kvantovací tabulky pro jasovou i rozdílovou barevnou složku z předem daných standardních tabulek (Obrázek 15), které zaručují faktor kvality q=50, který pro pozorování lidským okem plně dostačuje. Tyto kvantovací tabulky jsou navíc vynásobeny korelačními koeficienty pro 1-D AA&N DCT, aby se snížilo množství násobících operací při samotné části kompresního výpočtu. Po tomto kroku jsou sestaveny Huffmannovy tabulky z předem daných šablon ze standardu JPEG ([7] Annex K.3), a také je vytvořena tabulka diferenčních kategorií.

Jakmile jsou tyto tabulky nainicializovány, je možné přistoupit k vložení segmentů do výstupního souboru. Nejdříve je vložena značka začátku souboru SOI, za ní je vložen segment APP0, který označuje, že se jedná o soubor ve formátu JFIF. Následně jsou do výstupního souboru zapsány použité kvantizační tabulky DQT, které musí být seřazeny ZIGZAG řazením. Za nimi je zapsán segment se značkou SOF0, který specifikuje obrazová data (rozlišení, bitová hloubka, atd.). Potom následuje segment DHT obsahující Huffmannovy tabulky a nakonec je zapsán segment SOS, který upřesňuje formát obrazových dat a také označuje jejich začátek.

5.2.4 Převod barevného modelu, výpočet DCT a kvantizace koeficientů

Tato fáze výpočtu je implementovaná třemi způsoby, z řídící funkce `main()` je spuštěna taková varianta výpočtu, která odpovídá zadanému argumentu.

5.2.4.1 Sekvenční – jednovláknová implementace

V jednovláknové implementaci jsou tyto funkce realizovány v modulu `MyCPUFunc.cpp`. Podle počtu najednou zpracovávaných bloků je naalokována paměť a bloky jsou do ní z dočasného souboru načteny. Následně je postupně nad všemi bloky v paměti proveden převod barevného modelu, dále je nad každým jednotlivým blokem provedena diskrétní kosinová transformace v její optimalizované variantě 1-D AA&N DCT a potom je nad blokem provedena kvantizace. Nad takto zpracovanými

bloky se zavolá funkce pro Huffmannovo zakódování (viz sekce 5.2.5), a jakmile je zakódování hotovo, jsou ze vstupního souboru načítány nové bloky pro zpracování, a předcházející kroky se opakují, dokud nejsou zpracovány všechny bloky v dočasném souboru.

5.2.4.2 CUDA implementace

Funkce pro převod barevného modelu, DCT a kvantizace pomocí technologie CUDA jsou implementovány v modulu `MyGPUFuncCUDA.cu`. Při výpočtu pomocí grafické karty je důležité si nejprve zjistit informace o grafické kartě, na které bude výpočet probíhat. V ukázce kódu (Kód 3) je znázorněno, jakým způsobem je možné zjistit vlastnosti o aktuální grafické kartě. Po provedení tohoto kódu jsou ve struktuře `prop` obsaženy informace o maximálním počtu bloků (maxgrid), maximum vláken v bloku atd. Dále v proměnné `free` je velikost aktuální volné paměti. Z těchto informací můžeme určit, kolik budeme najednou zpracovávat obrazových bloků (viz sekce 5.2.2.2).

```
int device_id;
//získaní id aktuální používané grafiky
cudaGetDevice(&device_id);
//získaní vlastnosti dane grafiky
cudaDeviceProp prop;    //struktura pro informace o gpu
cudaGetDeviceProperties(&prop,device_id);
//velikost paměti
size_t free, total; // volna / obsazena
cudaMemGetInfo(&free, &total);
```

Kód 3: Zjištění id používaného CUDA zařízení a jeho vlastností (CUDA).

Jakmile máme zjištěno, kolik obrazových bloků budeme najednou zpracovávat, naalokujeme si paměťové objekty v grafické paměti. V následující části kódu (Kód 4) je znázorněno vytvoření paměťového objektu na grafické kartě a jeho naplnění daty.

```
float *lum;
//alokace grafické paměti o velikosti my_alloc_qt
cudaMalloc((void **)&lum, my_alloc_qt);
//zápis do grafické paměti
//kopírovaní probíhá z proměnné lumin_conv do lum
//počet zapisovaných bytu udává proměnná my_alloc_qt
cudaMemcpy(lum, lumin_conv, my_alloc_qt, cudaMemcpyHostToDevice);
```

Kód 4: Ukázka alokace a zápisu do grafické paměti na příkladu s kvantovací tabulkou (CUDA).

Následně je podle počtu najednou zpracovávaných obrazových bloků naalokován prostor v paměti RAM. Dále jsou do tohoto prostoru načteny bloky z dočasného souboru. Z operační paměti jsou tyto bloky překopírovány do naalokovaných paměťových objektů na grafické kartě. Dále je spuštěn převod barevného modelu (Kód 5), kde každý blok 8x8 pixelů zpracovává 64 vláken (1 pixel = 1 vlákno). V kódu (Kód 6) je zobrazena implementace kernelu, který převádí barvový model.

```
//global_item_size_color – počet cuda bloku, na kterých probíhá
výpočet
size_t global_item_size_color =
my_image.my_dib_header.blocks_count64/block;
//local_item_size_color – počet vláken v cuda bloku, na kterých
probíhá výpočet
size_t local_item_size_color = block;
//spuštěni kernelu na převod barev
//memR, memG, memB - vstupní paměťové objekty v RGB
//memY. memCb, memCr – výstupní paměťové objekty v YCbCr
color<<< global_item_size_color, local_item_size_color >>>
(memR, memG, memB, memY, memCb, memCr);
```

Kód 5: Spuštěni kernelu, který převádí barvové modely(CUDA).

```
//CUDA kernel pro převod barevného modelu
//threadIdx.x – index aktuálního vlákna v rámci bloku
//blockDim.x  – počet vláken v bloku
//blockIdx.x  – index aktuálního bloku
//i - globální jedinečný index do vstupního a výstupního pole
//Y_R, Y_G, Y_R atd. převodní konstanty
__global__ void color(unsigned char *memR, unsigned char *memG,
unsigned char *memB, short *memY, short *memCb, short *memCr)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  float R = (float)memR[i];
  float G = (float)memG[i];
  float B = (float)memB[i];

  memY[i]  = round(Y_R *R+Y_G *G+Y_B *B - 128);
  memCb[i] = round(Cb_R*R+Cb_G*G+Cb_B*B);
  memCr[i] = round(Cr_R*R+Cr_G*G+Cr_B*B);
}
```

Kód 6: Kernel, který obstarává převod barvového modelu (CUDA).

Po převedení barvového modelu je nad výstupními daty spuštěn výpočet DCT. Diskrétní kosinová transformace je implementována v její optimalizované formě 1-D AA&N DCT. Tudíž je možné zpracovávat každý obrazový blok (8x8 pixelů) 8mi vlákny. Nejprve jednotlivá vlákna zpracují řádky v obrazovém bloku a následně sloupce. Nad sloupci je přímo provedena kvantizace. Když je výpočet hotov, vypočtené hodnoty jsou překopírovány z grafické paměti do operační paměti, a nad těmito hodnotami je provedeno Huffmannovo zakódování (viz sekce 5.2.5). Následně se tento výpočet opakuje, dokud není dočasný soubor s obrazovými bloky prázdný.

Nakonec je potřeba dealokovat všechny použité paměťové objekty na grafické kartě (Kód 7).

```
//dealokace paměťového objektu
cudaFree(lum);
```

Kód 7: Ukázka dealokace paměťového objektu (CUDA).

5.2.4.3 OpenCL implementace

Implementace této vykonávací části v OpenCL je realizovaná ve dvou modulech. V prvním modulu (`MyGPUFundCL.cpp`), který je zároveň hostovský, jsou zpracovány řídící operace výpočtu, jako je

alokace paměťových objektů, přenosy dat mezi operační a grafickou pamětí a volání výpočetních kernelů. Druhým modulem je souboru `MyGDCT.cl`, ve kterém jsou funkce, jejichž běh je vykonávám na vybraném OpenCL zařízení. Tento druhý modul není překládán při samotné kompilaci aplikace, ale je nutné ho mít u výsledné binární formy aplikace, která si ho zkompiluje sama při použití OpenCL varianty výpočtu.

Samotný výpočet na OpenCL nejdříve začíná tím, že je nutné zjistit, zda se v počítači nachází nějaká OpenCL platforma. Následně je z platformy vybráno zařízení, na kterém bude výpočet probíhat a je vytvořen vykonávací kontext a příkazová fronta (Kód 8).

```
//zjištěni platformy
cl_platform_id *platform_id;
cl_uint ret_num_platforms;      //bude obsahovat počet možných
platforem
clGetPlatformIDs(1, platform_id, &ret_num_platforms);
//vybráni zařízeni z platformy – vybíráno výchozí zařízeni pro dany
systém
cl_device_id device_id;
cl_uint ret_num_devices;     //bude obsahovat počet možných zařízení
na platformě
clGetDeviceIDs(&platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
//vytvořeni vykonávacího kontextu na vybraném zařízeni
cl_context context;
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
//vytvořeni vykonávací příkazové fronty na vybraném zařízeni
cl_command_queue command_queue;
command_queue = clCreateCommandQueue(context,device_id, NULL, &err);
```

Kód 8: Ukázka kódu pro vytvoření vykonávacího prostředí pro OpenCL zařízení (OpenCL).

Následně je nutné si naalokovat potřebné paměťové prostory pro práci s daty na OpenCL zařízení. Velikost paměťových objektů odvodíme od počtu najednou zpracovávaných obrazových bloků (viz. sekce 5.2.2.3). Ukázka způsobu alokace a zápisu dat do paměťového objektu je zobrazena v následujícím kódu (Kód 9).

```
//Vytvoření paměťového objektu
cl_mem lum;
//Alokace paměti zařízeni
//CL_MEM_READ_ONLY – z pohledu zařízeni paměť pouze pro čteni
//my_alloc_qt – velikost alokované paměti v bytech
//err – obsahuje informaci o úspěchu či neúspěchu alokace
lum = clCreateBuffer(context, CL_MEM_READ_ONLY, my_alloc_qt, NULL,
&err);
//Zápis do paměťového objektu
//je zkopírováno my_alloc_qt bytů z pole lumin_conv, uloženého v
RAM paměti do paměti na OpenCL zařízení lum
clEnqueueWriteBuffer(command_queue, lum, CL_TRUE, 0, my_alloc_qt,
                     lumin_conv, 0, NULL, NULL);
```

Kód 9: Alokace a zápis do paměťového objektu na OpenCL zařízení (OpenCL).

Kernelům (funkcím vykonávaným na OpenCL zařízení) je nutné nastavit argumenty odlišným stylem, než jaký je znám z klasického C++ či NVIDIA CUDA C. Argumenty se danému kernelu přiřazují pomocí speciálního příkazu, jehož zápis můžeme vidět v následujícím úseku kódu (Kód 10). Takto se postupně přiřadí k danému kernelu všechny jeho argumenty.

```
//Nastavení argumentů funkce color
//color – je název samotného kernelu
//0 – označuje, že paměťový objekt memR je nultým (prvním) argumentem
daného kernelu
clSetKernelArg(color, 0, sizeof(cl_mem), (void *)&memR);
```

Kód 10: Nastavení argumentů kernelu (OpenCL).

Jakmile jsou vytvořeny a nainicializovány všechny potřebné paměťové objekty a všem kernelům jsou přiřazeny jejich argumenty, je možné přistoupit k samotnému převodu barvového modelu, DCT a kvantizace. Postupně jsou načítány obrazové bloky z dočasného souboru, dokud není načten jejich určený počet pro jeden cyklus výpočtu. Tyto načtené bloky jsou následně zkopírovány do paměťových objektů na OpenCL zařízení. Následně je spuštěn kernel pro převod barvového modelu, ve kterém je jednou pracovní jednotkou (1 vlákno) zpracováván jeden obrázkový pixel (viz Kód 11 a Kód 12).

Po převedení barvového modelu probíhá výpočet 1-D AA&N DCT, obdobně jako při implementaci v CUDA. Každý jednotlivý obrazový blok je zpracováván 8 pracovními jednotkami. Nejdříve jsou zpracovány řádky (1 řádek = 1 pracovní jednota) a následně sloupce. Na konci fáze výpočtu sloupců jsou bloky nakvantovány. Výsledné bloky jsou překopírovány do operační paměti, kde je nad nimi provedeno Huffmannovo zakódování (viz sekce 5.2.5). Takto jsou postupně zpracovány všechny obrazové bloky v dočasném souboru.

```
//Voláni kernelové funkce
//global_item_size_color – celkový počet všech pracovních jednotek,
které musí být spuštěny
size_t global_item_size_color =
my_image.my_dib_header.blocks_count64;
//local_item_size_color – počet pracovních jednotek v jedné pracovní
skupině
size_t local_item_size_color = block;
//volaní kenelu color
//command_queue – určeni fronty, kam bude výpočet zařazen
//color – výběr volané kernelové funkce
//1 – určuje v kolika dimenzích, budou pracovní jednotky spuštěny
clEnqueueNDRangeKernel(command_queue, color, 1, NULL,
&global_item_size_color, &local_item_size_color, 0, NULL, NULL);
```

Kód 11: Spuštění kernelové funkce v OpenCL (OpenCL).

```
//Kernel napsaný v OpenCL, pro převod barvoveho modelu
//před paměťovými objekty je nutné udávat identifikátor jejich
paměťového prostoru
//get_global_id(0) – obsahuje unikatni globalni index dane pracovní
jednotky
//Y_R, Y_G, Y_R atd. prevodni konstanty
__kernel void color(__global unsigned char *memR, __global unsigned
char *memG,
                    __global unsigned char *memB, __global short *memY,
                    __global short *memCb, __global short *memCr)
{
  __private int i = get_global_id(0);
  __private short R = memR[i];
  __private short G = memG[i];
  __private short B = memB[i];

  memY[i]  = round(Y_R *R+Y_G *G+Y_B *B - 128);
  memCb[i] = round(Cb_R*R+Cb_G*G+Cb_B*B);
  memCr[i] = round(Cr_R*R+Cr_G*G+Cr_B*B);
}
```

Kód 12: Ukázka kernelu napsaného v OpenCL, který má na starost převod barvy.

Když je výpočet dokončen, je nutné provést dealokaci paměťových objektů na OpenCL zařízení.

5.2.5 Huffmannovo zakódování

Nad obrazovými bloky, které již prošli fází převodu barvového modelu, DCT a kvantizací se provádí Huffmannovo zakódování. Funkce, která se stará o toho zakódování, je implementovaná v modulu `MySaver.cpp` a nazývá se `huffmann()` a jako parametr přijímá strukturu zpracovaných bloků všech třech barevných složek.

Samotné zakódování probíhá sekvenčně jeden blok po druhém v určeném pořadí. Nejdříve 1. blok Y, 1. blok C_b, 1. blok C_r, 2 blok Y atd.

Kódování samotného bloku probíhá tak, že nejdříve se vezme první hodnota, kterou je DC složka a od ní se odečte hodnota DC složky předešlého bloku stejné barvy. Pokud se jedná o první zakódovávaný blok, odečte se hodnota nula (viz sekce 4.2.6.1). Podle výsledné hodnoty, se určí diferenční kategorie, jejíž kód se podle Huffmannovy tabulky pro DC složky vloží do výstupního souboru. Potom je vložena odpovídající zástupná hodnota z tabulky diferenčních kategorií pro zakódovávanou hodnotu.

Následně jsou zpracovávány AC složky bloku, je zjištěno kolik nulových hodnot se nachází před každou nenulovou a podle toho je daná nenulová hodnota zakódována, obdobně jako při zakódovávání DC složek. Za poslední nenulovou hodnotou, pokud to není poslední značka v bloku, se vloží ukončovací značka bloku tzv. EOB a přistoupí se ke kódování následujícího bloku.

Podle toho, zda je zpracovávaný blok luminanční (jasový) či chrominanční (rozdílový) se musí použít odpovídající Huffmannovy AC či DC tabulky.

5.2.6 Uzavření výstupního obrázku a ukončení aplikace

Jakmile jsou zpracovány všechny bloky z dočasného souboru, je tento soubor uzavřen a odstraněn. Do výstupního souboru formátu JFIF je zapsán ukončující segment EOI. Postupně jsou dealokovány všechny použité paměťové proměnné a aplikace je ukončena.

6 Testování výkonu

Z důvodu, aby bylo možné nějakým racionálním způsobem vyhodnotit výkonnost jednotlivých variant výpočtu (převodu barvového modelu, DCT a kvantovací části) jsem do aplikace naimplementoval funkce pro měření času. V klasické C++ implementaci jsem využil schopností knihovny `<ctime>` a její funkce `clock()`.

Pro měření času stráveného při výpočtu pomocí technologie NVIDIA CUDA jsem použil její funkci tzv. `cudaEvent`, která je schopna měřit vykonávací čas na grafické kartě.

Aby bylo možné měřit spotřebovaný čas také při výpočtu pomocí OpenCL, je nutné u vykonávací příkazové fronty povolit funkci profilování. Tuto funkci je možné povolit tak, že při samotném vytváření této fronty jí nastavíme parametr `CL_QUEUE_PROFILING_ENABLE`. Následně je možné měřit čas pomocí OpenCL `event`ů, které oznamují ukončení provádění zadané operace a obsahují v sobě čas provádění.

U technologií NVIDIA CUDA a OpenCL do měření vykonávacího času zahrnuji i čas potřebný na nakopírování/čtení paměťových objektů do/z paměti zařízení, na kterém je výpočet prováděn, protože bez připočítání tohoto času by výpočet nebyl relevantní. Důvodem je to, že pokud bychom počítali pouze čas spotřebovaný samotným výpočtem, který by mohl být velice rychlý, pak by mohla nastat situace, kdy by čas spotřebovaný na přesuny mezi paměťmi byl velký, ale do spotřebovaného času by se nezapočítal, díky tomu by naměřené výkonnostní zrychlení bylo velké, ale reálné zrychlení by mohlo být nulové. Proto je potřeba pro porovnání výkonu brát celkový vykonávací čas, ke kterému při vykonávání pomocí CUDA či OpenCL patří i čas spotřebovaný na paměťové operace.

6.1 Testovací data

Vstupní obrázky, na kterých bylo prováděno testování, byly převzaty z [13]. Jsou to snímky planety země ve vysokém rozlišení. Obrázky byly původně v grafickém formátu PNG, a proto byly pro účely naší aplikace pomocí programu IrfanView [14] převedeny do formátu BMP. Pro prezentaci výsledků testování jsou použity časové údaje naměřené při komprimaci snímku pojmenovaného `world_2km.bmp` (tento obrázek je na přiloženém DVD). V následující tabulce (Tabulka 3) jsou obsaženy údaje o tomto obrázku ve formátu BMP, ale také o tomto obrázku ve formátu JPG, který byl vytvořen komprimací ve vyvinuté demonstrační aplikaci `jpeg_gpu`.

`world_2km.bmp`	Vstupní obrázek BMP	Výstupní obrázek JPEG/JFIF
Velikost souboru:	683438 kB	13492 kB
Rozlišení obrázku:	21600 x 10800	21600 x 10800
Bitová hloubka (na pixel):	24 b	24 b

Tabulka 3: Parametry obrázku, na kterém je prováděno testování výkonu.

6.2 Hardware použitý k testování

Pro testování byly použity dvě grafické karty od firmy NVIDIA, jejichž parametry jsou popsány v tabulce níže (Tabulka 4). Bohužel se mi nepodařilo sehnat na testy žádnou grafickou kartu od firmy AMD. Výhodou, že testování bylo prováděno na kartách NVIDIA je ta skutečnost, že na nich běží jak CUDA tak OpenCL, a proto je možné výsledky pro obě GPGPU technologie porovnávat.

Název grafické karty	Architektura / Verze výpočetního modelu	Výpočetních jednotek	Počet multiprocesorů (Compute unit)	Procesorů na multiprocesor (Processing elements)	Takt procesorů (Processing element)
Quadro FX880M	GT200 / 1.2	48	6	8	1210MHz
Quadro NVS5400M	Fermi GF108 / 2.1	96	2	48	1320MHz

Tabulka 4: Vlastnosti testovacích grafických karet.

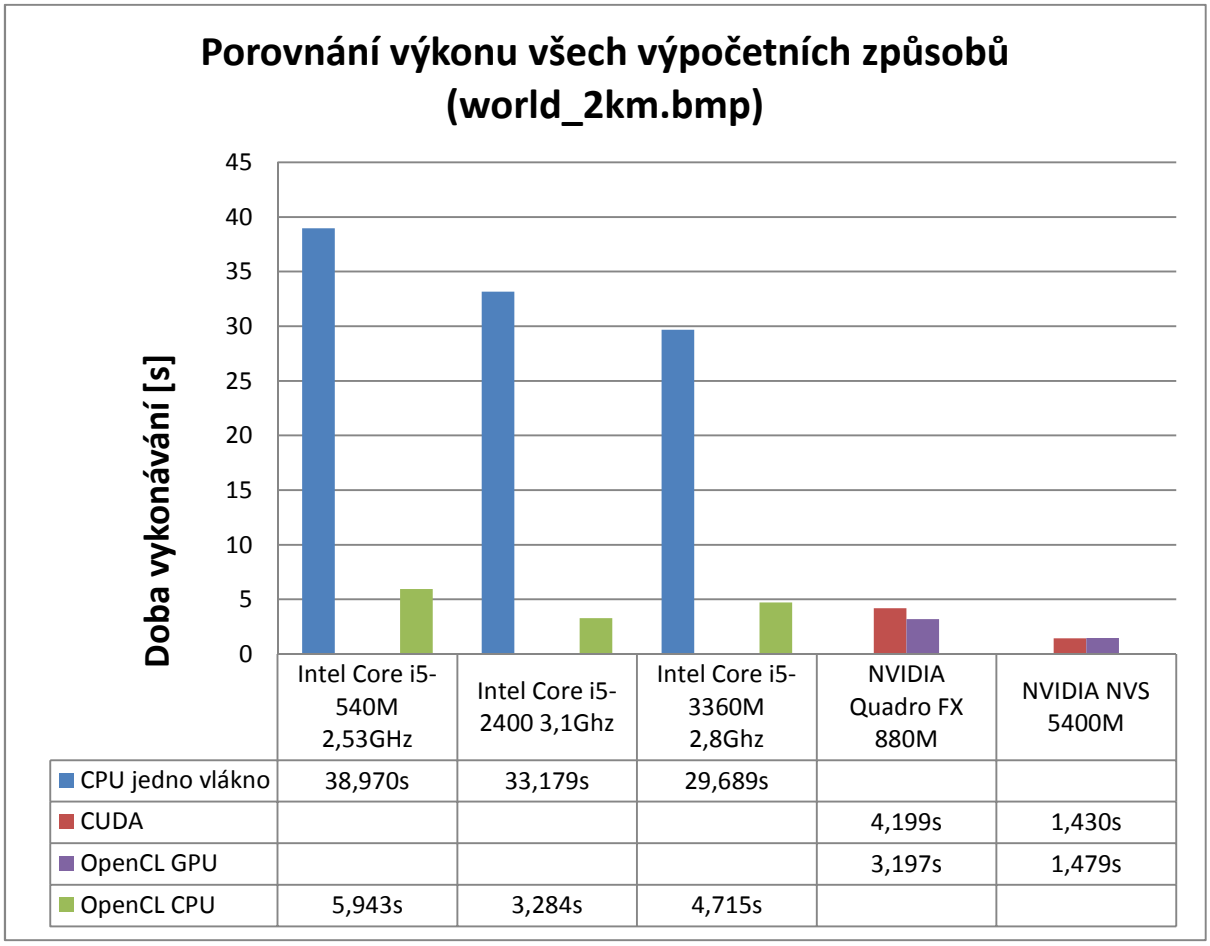
Vzhledem k tomu, že programový kód napsaný v OpenCL je možné vykonávat i na klasických procesorech, byl proveden test i na třech procesorech firmy Intel, jejichž parametry jsou uvedeny v tabulce (Tabulka 5). Při porovnání výkonu procesorů budeme porovnávat výkon mezi OpenCL implementací a implementací jednovláknovou v C++, která je vykonávána na jednom logickém jádře.

Název procesoru	Architektura	Fyzická jádra / Logická jádra (vlákna)	Takt	Možné přetaktování při využití pouze jednoho jádra
Intel Core i5-540M	Nehalem	2 / 4	2533MHz	3067MHz
Intel Core i5-2400	Sandy Bridge	4 / 4	3100MHz	3400MHz
Intel Core i5-3360M	Ivy Bridge	2 / 4	2800Mhz	3500MHz

Tabulka 5: Vlastnosti testovacích procesorů.

6.3 Výsledky testů při kompresi vstupního obrázku `world_2km.bmp`

V následujících testech byl měřen čas, potřebný k vykonání převodu barvového modelu, výpočtu diskrétní kosinové transformace, kvantizace a čas potřebný na paměťové přesuny u CUDA a OpenCL. Měření času bylo prováděno opakovaně a výsledné hodnoty byly získány zprůměrováním naměřených hodnot (cca 10 měření pro každou hodnotu). Na první obrázku (Obrázek 17) můžeme vidět výsledky naměřené všemi možnými způsoby na testovaném hardwaru pro daný obrázek. Z grafu a výsledků na obrázku je jasně patrné, že implementace pomocí technologií CUDA či OpenCL, výrazně urychlují výpočet oproti jednovláknové implementaci na CPU.

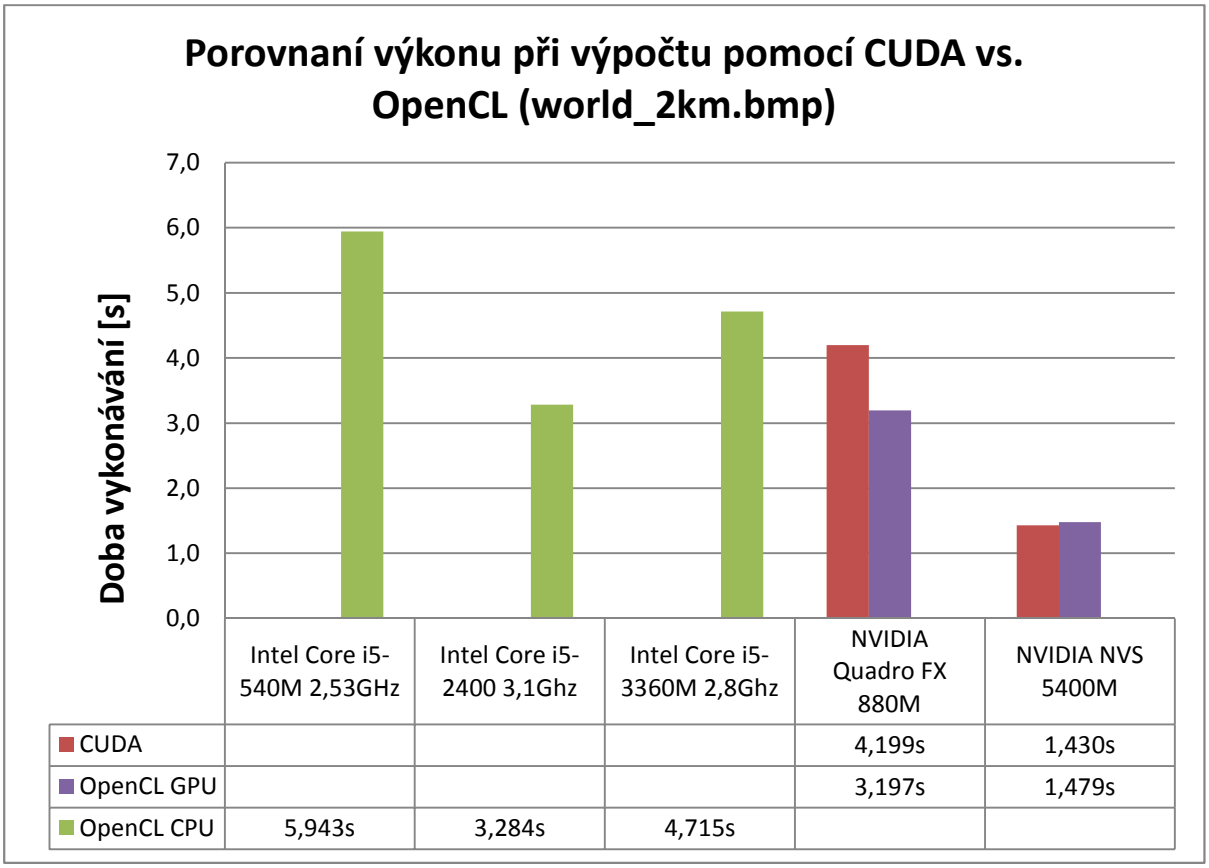


Obrázek 17: Porovnání výkonu všech výpočetních způsobů pro vstupní obrázek world_2km.bmp.

Na znázorněné jednojádrové implementaci (CPU jedno vlákno) si můžeme povšimnout, že hlavním parametrem, který určuje výkon daného procesoru je taktovací frekvence jádra. Musíme si ale uvědomit, že při sekvenční implementaci, kde je využíváno pouze jedno procesorové jádro, jsou procesory testovaných architektur schopny toto jádro mírně přetaktovat (viz. Tabulka 5). Díky této schopnosti je v sekvenční implementaci nejrychleší procesor Intel Core i5-3360M, který je schopen jedno své využívané jádro přetaktovat až na 3500MHz.

Následující graf (Obrázek 18), je shodný s předešlým grafem, ale je z něj odstraněna sekvenční jednojádrová implementace, která nám poněkud skrývala skutečné srovnání implementací pomocí CUDA či OpenCL.

Z grafu (Obrázek 18) je jasně patrné, že výpočet je nejrychlejší na grafické kartě Quadro NVS5400M, která disponuje 96 výpočetními jednotkami. Na této grafické kartě je výpočet pomocí OpenCL i CUDA v podstatě stejně rychlý, což ale neplatí u grafické karty Quadro FX880M, na které je výpočet pomocí technologie CUDA pomalejší přibližně o 30% oproti implementaci na OpenCL. Tato skutečnost je pravděpodobně způsobena tím, že grafické karty Quadro FX880M je postavena na starší grafické architektuře.



Obrázek 18: Porovnání výkonu implementací pomocí CUDA a OpenCL pro vstupní obrázek world_2km.bmp.

Také si můžeme prohlédnout výsledky naměřené pomocí technologie OpenCL na procesorech. Zde je vidět, že ačkoli mají všechny testované procesory stejný počet logických jader, jejich výkon se liší. Tato skutečnost je dána tím, že procesory Intel Core i5-540M a Intel Core i5-3360M disponují pouze 2 fyzickými jádry. Toho, že vlastní 4 logická jádra docilují pouze pomocí technologie Hyper Threading, která umožňuje běh 2 logických jader na jednom fyzickém. Jasným vítězem v porovnání procesorů je tedy Intel Core i5-2400, který sice nedisponuje technologií Hyper Threading, ale vlastní 4 fyzická výpočetní jádra a v této skutečnosti je jeho výpočetní síla.

Celkové testování probíhalo na více obrázcích, informace o všech obrázcích použitých při testování jsou obsaženy v Příloze 3. Do zobrazených výsledků jsem ale zahrnul pouze výsledky pro obrázek world_2km, který je největší z testovaných. Výsledky naměřené pro ostatní obrázky byly ve stejném výkonnostním poměru mezi zkoumanými výpočetními variantami, a proto je již neuvádím.

Z grafů je vidět, že výkon grafických karet (střední třídy) současných architektur je pro obě testované GPGPU technologie srovnatelný. Výkonově ale grafické karty převyšují klasické procesory, jak v OpenCL implementaci, tak v sekvenční jednojádrové implementaci, kde je rozdíl výkonu opravdu markantní.

7 Závěr

Cílem této bakalářské práce bylo zkoumání využitelnosti výpočetní síly grafických karet pro akceleraci kompresních algoritmů. Za zkoumaný kompresní algoritmus, na kterém byla tato využitelnost demonstrována, byl zvolen algoritmus pro ztrátovou kompresi obrázků JPEG.

Pro implementaci částí algoritmu JPEG, které lze realizovat paralelními výpočty byly pro možné porovnání zvoleny dvě GPGPU technologie, a to NVIDIA CUDA a OpenCL. Navíc pro porovnání se standardním procesorem byla ještě pro výpočet kompresních částí přidána jednovláknová implementace na procesoru, která znázorňuje výkon jednoho procesorového vlákna (jádra).

Všechny zmíněné výpočetní způsoby byly implementovány ve vyvinuté demonstrační aplikaci `jpeg_gpu`, která za pomoci kompresního algoritmu JPEG, převádí vstupní bitmapové obrázky (BMP) do formátu JPG.

Testováním pomocí demonstrační aplikace se zjistila výrazná výkonová výhoda realizace tohoto kompresního algoritmu pomocí grafické karty. Čas potřebný k výpočtu na grafických kartách byl menší, i přes nutnost přesunu dat mezi pamětí RAM a grafickou pamětí, což také stojí nějaký čas, který se při výpočtu pomocí procesoru nemusí uvažovat. Situace, kdy by čas vykonávání pomocí grafických karet nemusel být výhodný oproti procesorům, nastává u převodu malých obrázků, u kterých by se na GPU spouštěl malý počet výpočetních vláken. Tento problém, ale není potřeba řešit, protože u takto malých obrázků je jejich převodní čas skoro zanedbatelný.

Z mých zjištění, která vyplývají i z této bakalářské práce, si myslím, že zpracování paralelizovatelných algoritmů (ať již kompresních či jiných) pomocí grafických karet má v budoucnu veliký potenciál.

Na této práci si cením hlavně zkušeností, které jsem získal při programování data-paralelních grafických výpočetních architektur, které se do budoucna určitě budou dále rozšiřovat do různých odvětví informačních technologií.

Literatura

[1] NVIDIA CORPORATION. *NVIDIA CUDA C Programming Guide Version 5* [online]. 2012 [cit. 2013-04-11]. Dostupné z: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] BERILLO, Alexey. NVIDIA CUDA: Non-graphic computing with graphics processors. [online]. 2008 [cit. 2013-04-17]. Dostupné z: http://ixbtlabs.com/articles3/video/cuda-1-p5.html

[3] TSUCHIYAMA, Ryoji, Takashi NAKAMURA, Takuro IIZUKA, Akihiro ASAHARA, Jeongdo SON a Satoshi MIKI. *The OpenCL Programming Book* [online]. Fixstars, 2010 [cit. 2013-04-17]. Dostupné z: http://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/contents/

[4] KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification* [online]. 2011 [cit. 2013-04-17]. Dostupné z: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[5] TIŠNOVSKÝ, Pavel. Seriál Grafické formáty. [online]. 2008 [cit. 2013-04-23]. Dostupné z: http://www.root.cz/serialy/graficke-formaty/

[6] SALOMON, David. *Data Compression: The Complete Reference*. London: Springer Science+Business Media, 2007. ISBN 1-84628-602-6.

[7] INTERNATIONAL TELECOMMUNICATION UNION ITU. *Terminal Equipment and protocols for telematic services: Information Technology - Digital compression and coding of continuous-tome still images* . T. 81. [online]. 1993 [cit. 2013-04-25]. Dostupné z: http://www.w3.org/Graphics/JPEG/itu-t81.pdf

[8] KHRONOS GROUP: OpenCL. [online]. [cit. 2013-04-29]. Dostupné z: http://www.khronos.org/opencl/

[9] NVIDIA: CUDA Toolkit Documentation. [online]. [cit. 2013-04-29]. Dostupné z: http://docs.nvidia.com/cuda/

[10] PENNEBAKER, William B. a Joan L. MITCHELL. *JPEG: Still Image Data Compression Standard*. Norwell, Massachusetts USA: KLUVER ACADEMIC PUBLICHERS, 2004. ISBN 0-442-01272-1.

[11] BMP file format. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2013 [cit. 2013-04-30]. Dostupné z: http://en.wikipedia.org/wiki/BMP_file_format

[12] Description of IDCT Algorithm. MOONEY, Vincent J. [online]. 2007 [cit. 2013-04-30]. Dostupné z: http://www.prism.gatech.edu/~aaga3/ece6132/labs/lab2/idct.html

[13] Unearthed Outdoors: True Marble. [online]. [cit. 2013-05-09]. Dostupné z: http://www.unearthedoutdoors.net/global_data/true_marble/download

[14] IrfanView. [online]. [cit. 2013-05-09]. Dostupné z: http://www.irfanview.cz/

Seznam příloh

Příloha 1. Popis demonstrační aplikace `jpeg_gpu`

Přiložené DVD obsahuje, jak zdrojové kódy vytvářené aplikace, tak její přeloženou binární formu. Aplikace je řízena podle vstupních argumentů. První argument určuje způsob vykonávání komprimačního algoritmu JPEG. Druhým parametrem obrázek ve formátu BMP. Výstupem z programu je obrázek ve formátu JPEG/JFIF, který je uložen ve stejném umístění jako vstupní obrázek i se shodným názvem, ale má koncovku `.jpg`.

Jako první argument mohou být použity tyto přepínače:

- `--def` – spustí konverzi obrázku, která ale nevyužívá možností GPGPU, je napsána pomocí C++ (využívá tudíž pouze jedno procesorové vlákno/jádro).
- `--cuda` – pokud je v počítači přítomna grafická karta od firmy NVIDIA, která podporuje technologii CUDA, je výpočet akcelerován pomocí této technologie. Samotné sekvenční části kompresního programu jsou napsány v jazyce CUDA C.
- `--clgpu` – je-li v počítači přítomna OpenCL kompatibilní grafická karta je výpočet spuštěn na ní. Samotná implementace je provedena v OpenCL C.
- `--clcpu` – tento parametr je ve své podstatě shodný s předchozím parametrem, ale protože programy, které jsou implementovány pomocí OpenCL je možné spouštět i na klasických procesorech, které ale musejí mít podporu OpenCL, je výpočet spuštěn na CPU. Ostatní parametry jsou shodné s parametrem --clgpu.
- `--help` – za tento parametr se již nic nepřidává, a je vypsána programová nápověda.

Příklady spuštění aplikace:

```
jpeg_gpu.exe –-clgpu world_2km.bmp
jpeg_gpu.exe –-cuda  world_2km.bmp
jpeg_gpu.exe --help
```

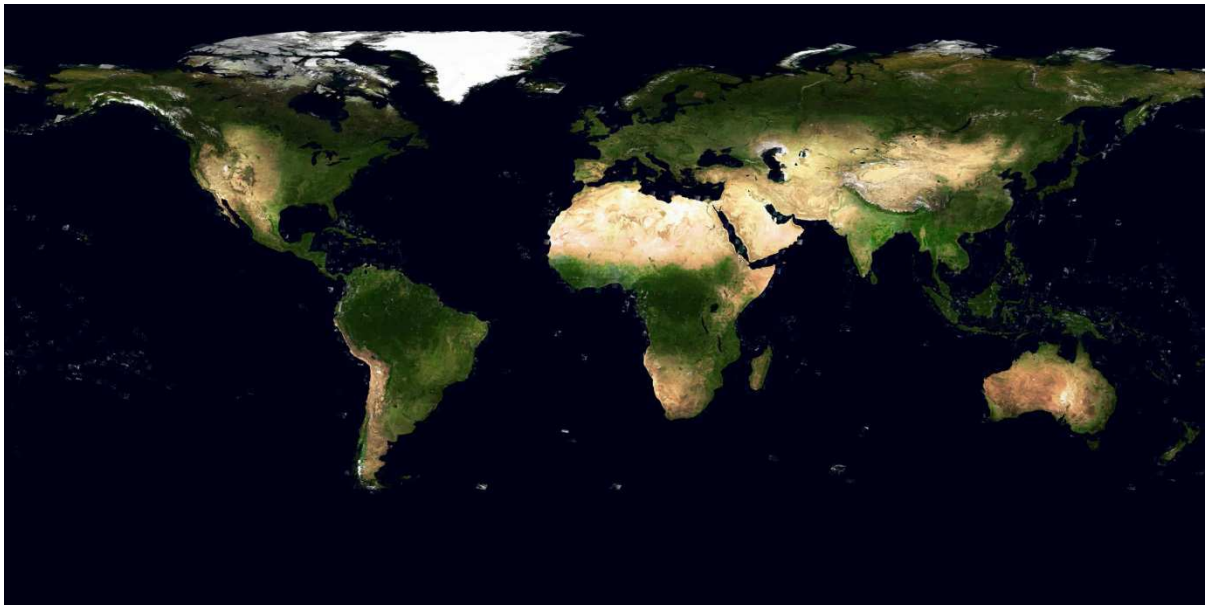
Příloha 2. Obsah DVD

Obsah přiloženého DVD je rozdělen do této adresářové struktury:

- *bin* – obsahuje přeložený program jpeg_gpu.exe se všemi potřebnými knihovnami ke spuštění
- *src* – obsahuje zdrojové soubory aplikace z programu Microsoft Visual Studio 2010
- *images_in* – v této složce jsou uloženy testovací vstupní obrázky ve formátu BMP
- *images_out* – zde jsou pro ukázku uloženy ekvivalentní obrázky ve formátu JPEG/JFIF, ke vstupním obrázkům, vytvořené vyvinutou aplikací
- *text_pdf* – obsahuje tento text ve formátu PDF
- *text_src* – obsahuje zdrojový formát tohoto textu ve formátu Microsoft Word (DOCX)
- *manual* – složka obsahuje manuál, ve kterém je popsán způsob překladu implementované aplikace

Příloha 3. Ukázka a popis testovacích obrázků

Zde je zobrazen testovací obrázek `world_16km.jpg` (Příloha 3. Obrázek 1), který vznikl kompresí vstupního obrázku `world_16km.bmp`. Tento obrázek je vizuálně shodný s obrázkem, který byl použit pro měření výkonu v části 6.3, pouze má tento obrázek nižší rozlišení, a proto je vhodnější pro vložení do testu práce.



Příloha 3. Obrázek 1: Ukázka obrázku world_16km.jpg.

V následující tabulce (Příloha 3. Tabulka 1) jsou zobrazeny informace o všech obrázcích, které jsem využíval při porovnávání a testování výkonu aplikace. Výstupní soubor JPEG/JFIF je vytvořen vyvinutou demonstrační aplikací. Všechny tyto obrázky jsou obsaženy na přiloženém disku DVD.

Název	Rozlišení	Bitová hloubka (na pixel):	Velikost vstupního BMP	Velikost výstupního JPEG/JFIF
world_16km	2700x1350	24b	10679kB	238kB
world_8km	5400x2700	24b	42715kB	1054kB
world_4km	10800x5400	24b	170860kB	3189kB
world_2km	21600x10800	24b	683438kB	13492kB

Příloha 3. Tabulka 1: Tabulka obsahující informace o všech obrázcích použitých při testování.